Noname manuscript No.
(will be inserted by the editor)

Distant decimals of π

Formal proofs of some algorithms computing them and guarantees of exact computation

Yves Bertot · Laurence Rideau · Laurent Théry

Received: date / Accepted: date

Abstract We describe how to compute very far decimals of π and how to provide formal guarantees that the decimals we compute are correct. In particular, we report on an experiment where 1 million decimals of π and the billionth hexadecimal (without the preceding ones) have been computed in a formally verified way. Three methods have been studied, the first one relying on a spigot formula to obtain at a reasonable cost only one distant digit (more precisely a hexadecimal digit, because the numeration basis is 16) and the other two relying on *arithmetic-geometric means*. All proofs and computations can be made inside the Coq system. We detail the new formalized material that was necessary for this achievement and the techniques employed to guarantee the accuracy of the computed digits, in spite of the necessity to work with fixed precision numerical computation.

Keywords Formal proofs in real analysis · Coq proof assistant · Arithmetic Geometric Means · Bailey & Borwein & Plouffe formula · BBP · PI

1 Introduction

The number π has been exciting the curiosity of mathematicians for centuries. Ingenious formulas to compute this number manually were devised since antiquity with Archimede's exhaustion method and a notable step forward achieved in the eighteenth century, when John Machin devised the famous formula he used to compute one hundred decimals of π.

This work was partially funded by the ANR project *FastRelax*(ANR-14-CE25-0018-01) of the French National Agency for Research

Yves Bertot, Laurence Rideau, Laurent Théry
Inria Sophia Antipolis
Université Côte d'Azur
E-mail: firstname.name@inria.fr

Today, thanks to electronic computers, the representation of π in fractional notation is known up to tens of trillions of decimal digits. Establishing such records raises some questions. How do we know that the digits computed by the record-setting algorithms are correct? The accepted approach is to perform two computations using two different algorithms. In particular, with the help of a spigot formula, it is possible to perform a statistical verification, simply checking that a few randomly spread digits are computed correctly.

In this article, we study the best known spigot formula, an algorithm able to compute a faraway digit at a cost that is much lower than computing all the digits up to that position. We also study two algorithms based on arithmetic geometric means, which are based on iterations that double the number of digits known at each step. For these algorithms, we perform all the proofs in real analysis that show that they do converge towards π, giving the rate of convergence, and we then show that all the computations in a framework of fixed precision computations, where computations are only approximated by rational numbers with a fixed denominator, are indeed correct, with a formally proved bound on the difference between the result and π. Last we show how we implement the computations in the framework of our theorem prover.

The first algorithm, due to Bailey, Borwein, and Plouffe relies on a formula of the following shape, known as the BBP formula [4].

$$\pi = \sum_{i=0}^{\infty} \frac{1}{16^i} \left(\frac{4}{8i+1} - \frac{2}{8i+4} - \frac{1}{8i+5} - \frac{1}{8i+6} \right).$$

Because each term of the sum is multiplied by $\frac{1}{16^i}$ it appears that approximately n terms of the infinite sum are needed to compute the value of the nth hexadecimal digit. Moreover, if we are only interested in the value of the nth digit, the sum of terms can be partitioned in two parts, where the first contains the terms such that $i \leq n$ and the second contains terms that will only contribute when carries need to be propagated.

We shall describe how this algorithm is proved correct and what techniques are used to make this algorithm run inside the Coq theorem prover.

The second and third algorithms rely on a process known as the *arithmetic-geometric mean*. This process considers two inputs a and b and successively computes two sequences a_n and b_n such that $a_0 = a$, $b_0 = b$, and

$$a_{n+1} = \frac{a_n + b_n}{2} \qquad b_{n+1} = \sqrt{a_n b_n}$$

In the particular case where $a = 1$ and $b = x$, the values a_n and b_n are functions of x that are easily shown to be continuous and differentiable and it is useful to consider the two functions

$$y_n(x) = \frac{a_n(x)}{b_n(x)} \qquad z_n = \frac{b'_n(x)}{a'_n(x)}$$

A first computation of π is expressed by the following equality:

$$\pi = (2 + \sqrt{2}) \prod_{n=1}^{\infty} \frac{1 + y_n(\frac{1}{\sqrt{2}})}{1 + z_n(\frac{1}{\sqrt{2}})}.$$

Truncations of this infinite product are shown to approximate π with a number of decimals that doubles every time a factor is added. This is the basis for the second algorithm.

The third algorithm also uses the arithmetic geometric mean for 1 and $\frac{1}{\sqrt{2}}$, but performs a sum and a single division:

$$\pi = \lim_{n\to\infty} \frac{4(a_n(1, \frac{1}{\sqrt{2}}))^2}{1 - \sum_{i=1}^{n-1} 2^{i-1}(a_{i-1}(1, \frac{1}{\sqrt{2}}) - b_{i-1}(1, \frac{1}{\sqrt{2}}))^2}$$

It is sensible to use index n in the numerator and $n-1$ in the sum of the denominator, because this gives approximations with comparable precisions of their respective limits. This is the basis for the third algorithm. This third algorithm was introduced in 1976 independently by Brent and Salamin [14,37]. It is the one implemented in the `mpfr` library for high-precision computation [21] to compute π.

In this paper, we will recapitulate the mathematical proofs of these algorithms (sections 2 and 3), and show what parts of existing libraries of real analysis we were able to reuse and what parts we needed to extend.

For each of the algorithms, we study first the mathematical foundations, then we concentrate on implementations where all computations are done with a single-precision fixed-point arithmetic, which amounts to forcing all intermediate results to be rational numbers with a common denominator. This framework imposes that we perform more proofs concerning bounds on accumulated rounding errors.

Context of this work. All the work described in this paper was done using the Coq proof assistant [19]. This system provides a library describing the basic definition of real analysis, known as *the standard Coq library for reals*, where the existence of the type of real numbers as an ordered, archimedian, and complete field with decidable comparison is assumed. This choice of foundation makes that mathematics based on this library is inherently classical, and real numbers are abstract values which cannot be exploited in the programming language that comes in Coq's type theory.

The standard Coq library for reals provides notions like convergent sequences, series, power series, integrals, and derivatives. In particular, the sine and cosine functions are defined as power series, π is defined as twice the first positive root of the cosine function, and the library provides a first approximation of $\frac{\pi}{2}$ as being between $\frac{7}{8}$ and $\frac{7}{4}$. It also provides a formal description of Machin formulas, relating computation of π to a variety of computations of arctangent at rational arguments, so that it is already possible to compute relatively close approximations of π, as illustrated in [6].

The standard Coq library implements principles that were designed at the end of the 1990s, where values whose existence is questionable should always be guarded by a proof of existence. These principles turned out to be impractical for ambitious formalized mathematics in real analysis, and a new library called Coquelicot [11] was designed to extend the standard Coq library and achieve a

more friendly and regular interface for most of the concepts, especially limits, derivatives, and integrals. The developments described in this paper rely on Coquelicot.

Many of the intermediate level steps of these proofs are performed automatically. The important parts of our working context in this respect are the `Psatz` library, especially the `psatzl` tactic [9], which solves reliably all questions that can be described as linear arithmetic problems in real numbers and `lia` [9], which solves similar problems in integers and natural numbers. Another tool that was used more and more intensively during the development of our formal proofs is the `interval` tactic [34], which uses interval arithmetic to prove bounds on mathematical formulas of intermediate complexity. Incidentally, the `interval` tactic also provides a simple way to prove that π belongs to an interval with rational coefficients.

Intensive computations are performed using a library for computing with very large integers, called `BigZ` [24]. It is quite notable that this library contains an implementation of an optimized algorithm to compute square roots of large integers [7].

2 The BBP formula

In this section we first recapitulate the main mathematical formula that makes it possible to compute a single hexadecimal at a low cost [4].

Then, we describe an implementation of an algorithm that performs the relevant computation and can be run directly inside the Coq theorem prover.

2.1 Proof of the BBP formula

2.1.1 The mathematical Proof

We give here a detailed proof of the formula established by David Bayley, Peter Borwein and Simon Plouffe. The level of detail is chosen to mirror the difficulties encountered in the formalization work.

$$\pi = \sum_{i=0}^{\infty} \frac{1}{16^i}\left(\frac{4}{8i+1} - \frac{2}{8i+4} - \frac{1}{8i+5} - \frac{1}{8i+6}\right) \tag{1}$$

We first study the properties of the sum S_k for a given k such that $1 < k$:

$$S_k = \sum_{i=0}^{\infty} \frac{1}{16^i(8i+k)} \tag{2}$$

By using the notation $[f(x)]_0^y = f(y) - f(0)$ and the laws of integration, we get

$$S_k = \sqrt{2}^k \sum_{i=0}^{\infty} \left[\frac{x^{k+8i}}{8i+k}\right]_0^{\frac{1}{\sqrt{2}}} = \sqrt{2}^k \sum_{i=0}^{\infty} \int_0^{\frac{1}{\sqrt{2}}} x^{k-1+8i}\,\mathrm{d}x \tag{3}$$

Thanks to uniform convergence, the series and the integral can be exchanged and we can then factor out x^{k-1} and recognize a geometric series in x^8.

$$S_k = \sqrt{2}^k \int_0^{\frac{1}{\sqrt{2}}} \sum_{i=0}^{\infty} x^{k-1+8i}\,\mathrm{d}x = \sqrt{2}^k \int_0^{\frac{1}{\sqrt{2}}} \frac{x^{k-1}}{1-x^8}\,\mathrm{d}x \tag{4}$$

Now replacing the S_k values in the right hand side of (1), we get:

$$S = 4S_1 - 2S_4 - S_5 - S_6 = \int_0^{\frac{1}{\sqrt{2}}} \frac{4\sqrt{2} - 8x^3 - 4\sqrt{2}x^4 - 8x^5}{1-x^8}\,\mathrm{d}x \tag{5}$$

Then, with the variable change $y = \sqrt{2}x$ and algebraic calculations on the integrand

$$S = \int_0^1 \frac{4-4y}{y^2-2y+2} + \frac{4}{1+(y-1)^2} + 4\frac{y}{y^2-2}\,\mathrm{d}y \tag{6}$$

We recognize here the respective derivatives of $-2\ln(y^2-2y+2)$, $4\arctan(y-1)$ and $2\ln(2-y^2)$. Most of these functions have null or compensating values at the bounds of the integral, leaving only one interesting term:

$$\begin{aligned} S &= \left[-2\ln(y^2-2y+2) + 4\arctan(y-1) + 2\ln(2-y^2)\right]_0^1 \\ &= -4\arctan(-1) = \pi \end{aligned}$$

2.1.2 The formalization of the proof

The current version of our formal proof, compatible with Coq version 8.5 and 8.6 [19] is available on the world-wide web [8]. To formalize this proof, we use the Coquelicot library intensively. This library deals with series, power series, integrals and provides some theorems linking these notions that we need for our proof. In Coquelicot, series (named `Series`) are defined as in standard mathematics as the sum of the terms of an infinite sequence (of type $nat \to R$ in our case) and power series (`PSeries`) are the series of terms of the form $a_n x^n$. The beginning of the formalisation follows the proof (steps (2) to (3)). Then, one of the key arguments of the proof is the exchange of the integral sign and the series allowing the transition from equation (3) to equation (4). The corresponding theorem provided by Coquelicot is the following:

```
Lemma RInt_PSeries (a : nat -> R) (x : R) :
   Rbar_lt (Rabs x) (CV_radius a) ->
   RInt (PSeries a) 0 x = PSeries (PS_Int a) x.
```

where `(PSeries (PS_Int` a`))` is the series whose (n+1)-th term is $\frac{a_n}{n+1}x^{n+1}$ coming from the equality: $\int_0^x a_n x^n = \left[\frac{a_n}{n+1}x^{n+1}\right]_0^x$. We use this lemma as a rewriting rule from right to left.

Note that the `RInt_PSeries` theorem assumes that the integrated function is a power series (not a simple series), that is, a series whose terms have the form $a_i x^i$. In our case, the term of the series is x^{k-1+8i}, that is $x^{k-1}x^{8i}$. To transform it into an equivalent power series we have first to transform the series $\sum_i x^{8i}$ into a power series. For that purpose, we define the `hole` function.

```
Definition hole (n : nat) (a : nat -> R) (i : nat) :=
  if n mod k =? 0  then a (i / n)  else 0.
```

and prove the equality given in the following lemma.

```
Lemma fill_holes k a x :
  k <> 0 -> ex_pseries a (x ^ k) ->
  PSeries (hole k a) x = Series (fun n => a n * x ^ (k * n)).
```

The premise written in the second line of `fill_holes` expresses that the series $\sum_i a_i(x^k)^i$ converges. This equality expresses that the series of term $a_i(x^k)^i$ is equivalent to the power series which terms are $a_{n/k}$ when n is a multiple of k and 0 otherwise.

Then by combining `fill_holes` with the Coquelicot function `(PS_incr_n a n)`, that shifts the coefficients of the series $\sum_{i=0}^{\infty} a_i x^{n+i}$ to transform it into $\sum_{i=0}^{i=n-1} 0.x^i + \sum_{i=n}^{\infty} a_{i-n}x^i$ that is a power series, we prove the `PSeries_hole` lemma.

```
Lemma PSeries_hole x a d k :
  0 <= x < 1 ->
  Series (fun i : nat => a * x ^ (d + S n * i)) =
  PSeries (PS_incr_n (hole (S k) (fun _ : nat => a)) n) x
```

Moreover, the `RInt_PSeries` theorem contains the hypothesis that the absolute value of the upper bound of the integral, that is $|x|$, is less than the radius of convergence of the power series associated to a. This is proved in the following lemma.

```
Lemma PS_cv x a :
  (forall n : nat, 0 <= a n <= 1) ->
  0 <= x -> x < 1 -> Rbar_lt (Rabs x) (CV_radius a)
```

It should be noted that in our case a_n is either 1 or 0 and the hypothesis `forall n : nat, 0 <= a n <= 1` is easily satisfied.

In summary, the first part of the proof is formalized by the `Sk_Rint` lemma:

```
Lemma Sk_Rint k (a := fun i => / (16 ^ i * (8 * i + k))) :
  0 < k ->
  Series a =
   sqrt 2 ^ k  *
       RInt (fun x => x ^ (n - 1) / (1 - x ^ 8)) 0 (/ sqrt 2).
```

that computes the value of S_k given by (4) from the definition (2) of `Sk`.

The remaining of the formalized proof follows closely the mathematical proof described in the previous section. We first perform an integration by substitution (starting from equation (5)), replacing the variable x by $\sqrt{2}x$, by rewriting (from right to left) with the `RInt_comp_lin` Coquelicot lemma.

```
Lemma RInt_comp_lin f u v a b :
  RInt (fun y : R => u * f (u * y + v)) a b =
  RInt f (u * a + v) (u * b + v)
```

This lemma assumes that the substitution function is a linear function, which is the case here.
Then we decompose `S` into three parts (by computation) to obtain equation (6), actually decomposed into three integrals that are computed in lemmas `RInt_Spart1`, `RInt_Spart2`, and `RInt_Spart3` respectively. For instance:

```
Lemma RInt_Spart3 :
  RInt (fun x => (4 * x) / (x ^ 2 - 2)) 0 1 = 2 * (ln 1 - ln 2).
```

Finally, we obtain the final result, based on the equality $\arctan 1 = \frac{\pi}{4}$.

2.2 Computing the nth decimal of π using the Plouffe formula

We now describe how the formula (1) can be used to compute a particular decimal of π effectively. This formula is a summation of four terms where each term has the form $1/16^i(8i+k)$ for some k. Digits are then expressed in hexadecimal (base 16). Natural numbers strictly less than 2^p are used to simulate a modular arithmetic with p bits, where p is the precision of computation. We first explain how the computation of $S_k = \sum_i 1/16^i(8i+k)$ for a given k is performed. Then, we describe how the four computations are combined to get the final digit.

We want to get the digit at position d. The first operation is to scale the sum S_k by a factor $m = 16^{d-1}\,2^p$ to be able to use integer arithmetic. In what follows, we need that p is greater than four. If we consider $\lfloor mS_k \rfloor$ (the integer part of mS_k), the digit we are looking for is composed of its bits p, $p-1$, $p-2$, $p-3$ that can be computed using basic integer operations: $(\lfloor mS_k \rfloor \bmod 2^p)/2^{p-4}$. Using integer arithmetic, we are going to compute an approximation of $\lfloor mS_k \rfloor \bmod 2^p$ by splitting the sum into three parts

$$mS_k = \sum_{0 \le i < d} \frac{m}{16^i(8i+k)} + \sum_{d \le i < d+p/4} \frac{m}{16^i(8i+k)} + \sum_{d+p/4 \le i} \frac{m}{16^i(8i+k)} \quad (7)$$

In the first part, the inner term can be rewritten as $\frac{2^p 16^{d-1-i}}{8i+k}$ where both divisor and dividend are natural numbers. The division can be performed in several stages. To understand this, it is worth comparing the fractional and integer part of $\frac{16^{d-1-i}}{8i+k}$ with the bits of $\frac{2^p 16^{d-1-i}}{8i+k}$.

For illustration, let us consider the case where $i = 0$, $k = 3$, $p = 4$, and $d = 2$. The number we wish to compute is

$$\frac{2^4 16^{2-1}}{3}$$

and we only need to know the first 4 bits, that is we need to know this number modulo 2^4. The ratio is $85.33\overline{3}$, and modulo 16 this is 5. Now, we can look at the number $2^4 \frac{16}{3}$. If we note q and r the quotient and the remainder of the division on the left (when viewed as an integer division), we have

$$2^4 \frac{16}{3} = 2^4 q + \frac{2^4 r}{3}$$

Since we eventually want to take this number modulo 2^4, the left part of the sum, $2^4 q$, does not impact the result and we only need to compute r, in other words $16 \bmod 3$. In our illustration case, we have $16 \bmod 3 = 1$ and $\frac{2^4 \times 1}{3} = 5.333$, so we do recover the right 4 bits. Also, because we are only interested in bits that are part of the integral part of the result, we can use integer division to perform the last operation.

These computations are performed in the following Coq function, that progresses by modifying a state datatype containing the current index and the current sum. In this function, we also take care of keeping the sum under 2^p, because we are only concerned with this sum modulo 2^p.

```
Inductive NstateF := NStateF (i : nat) (res : nat).
```

Doing an iteration is performed by

```
Definition NiterF k (st : NstateF) :=
  let (i, res) := st in
  let r := 8 * i + k in
  let res := res + (2 ^ p * (16 ^ (d - 1 - i) mod r)) / r in
  let res := if res < 2 ^ p then res else res - 2 ^ p in
  NStateF (i + 1) res.
```

The summation is performed by d iterations:

```
Definition NiterL k := iter d  (NiterF k)  (NStateF 0 0).
```

The result of `NiterL` is a natural number. What we need to prove is that it is a modular result and it is not so far from the real value. As we have turned an exact division into a division over natural numbers, the error is at most 1. After d iterations, it is at most d. This is stated by the following lemma.

```
Lemma sumLE k (f := fun i => ((16 ^ d / 16) * 2 ^ p) /
                             (16 ^ i * (8 * i + k))) :
  0 < k ->
  let (_, res) := NiterL p d k in
  exists u : nat, 0 <= sum_f_R0 f (d - 1) - res - u * 2 ^ p < d.
```

where `sum_f_R0 f n` represents the summation $f(0) + f(1) + \ldots + f(n)$.

Let us now turn our attention to the second part of the iteration of formula (7).

$$\sum_{d \leq i < d+p/4} \frac{m}{16^i(8i+k)} = \sum_{d \leq i < d+p/4} \frac{2^p 16^{d-1-i}}{8i+k}.$$

All the terms of this sum are less than 2^p. As terms get smaller by a factor of at least 16, we consider only $p/4$ terms. We first build a datatype that contains the current index, the current shift and the current result:

```
Inductive NstateG := NStateG (i : nat) (s : nat) (res : nat).
```

We then define what is a step:

```
Definition NiterG k (st : NstateG) :=
  let (i, s, res) := st in
  let r := 8 * i + k in
  let res := res + (s / r) in
  NStateG (i + 1) (s / 16) res.
```

and we iterate $p/4$ times:

```
Definition NiterR k :=
  iter (p / 4)  (NiterG k) (NStateG d (2 ^ (p - 4)) 0).
```

Here we do not need any modulo since the result fits in p bits and as the contribution of each iteration makes an error of at most one unit with the division by r, the total error is then bounded by $p/4$. This is stated by the following lemma.

```
Lemma sumRE k (f := fun i =>
                      ((16 ^ d / 16) * 2 ^ p) /
                       (16 ^ (d + i) * (8 * (d + i) + k))) :
  0 < k -> 0 < p / 4 ->
  let (_, _, s1) := NiterR k in
  0 <= sum_f_R0 f (p / 4 - 1) - s1 < p / 4.
```

The last summation is even simpler. We do not need to perform any computation. all the terms are smaller than 1 and quickly decreasing. It is then easy to prove that this summation is strictly smaller than 1.

Adding the two computations, we get our approximation.

```
Definition NsumV k :=
  let (_, res1) := NiterL k in
  let (_, _, res2) := NiterR k in res1 + res2.
```

We know that it is an under approximation and the error is less than $d+p/4+1$.

We are now ready to define our function that extracts the digit:

```
Definition NpiDigit :=
  let delta := d + p / 4 + 1 in
  if (3 < p) then
    if 8 * delta < 2 ^ (p - 4) then
      let Y := 4 * (NsumV 1) +
             (9 * 2^ p -
             (2 * NsumV 4 + NsumV 5 + NsumV 6 + 4 * delta)) in
      let v1 := (Y + 8 * delta) mod 2 ^ p / 2 ^ (p - 4) in
      let v2 := Y mod 2 ^ p / 2 ^ (p - 4) in
      if v1 = v2 then Some v2 else None
    else None
  else None.
```

This deserves a few comments. In this function, the variable `delta` represents the error that is done by one application of `NsumV`. When adding the different sums, we are then going to make an overall error of `8 * delta`. Moreover,

we know that `NsumV` is an under approximation. The variable `Y` computes an under approximation of the result: for those sums that appear negatively, the under approximation is obtained adding `delta` to the sum before taking the opposite. This explains the fragment `... + 4 * delta` that appears on the seventh line. Each of the sums obtained by `NsumV` actually is a natural number s smaller than 2^p, when it is multiplied by a negative coefficient, this should be represented by $2^p - s$. Accumulating all the compensating instances of 2^p leads to the fragment `9 * 2 ^ p - ...` that appears on the sixth line.

After all these computations, `Y + 8 * delta` is an over approximation. If both `Y` and `Y + 8 * delta` give the same digit, we are sure that this digit is valid.

The correctness of the `NpiDigit` function is proved with respect to the definition of what is the digit at place d in base b of a real number r, i.e. we take the integer part of rb^d and we take the modulo b:

```
Definition Rdigit (b : nat) (d : nat) (r : R) :=
  (Int_part ((Rabs r) * (b ^ d))) mod b.
```

The correctness is simply stated as

```
Lemma NpiDigit_correct k :
  NpiDigit = Some k -> Rdigit 16 d PI = k.
```

Note that this is a partial correctness statement. A program that always returns `None` also satisfies this statement. If we look at the actual program, it is clear that one can precompute a p that fulfills the first two tests, the equality test is another story. A long sequence of 0 (or F) may require a very high precision.

This program is executable but almost useless since it is based on a Peano representation of the natural numbers. Our next step was to derive an equivalent program using a more efficient representation of natural numbers, provided by the type `BigN` [24]. This code also receives some optimizations to implement faster operations of multiplications and divisions by powers of 2 and fast modular exponentiations.

Computing within Coq that 2 is the millionth decimal in hexadecimal of π with a precision of 28 bits (27 are required for the first two tests and 28 for the equality test) takes less than 2 minutes. In order to reach the billionth decimal, we implement a very naive parallelization for a machine with at least four cores: each sum is computed on a different core generating a theorem then the final result is computed using these four theorems. With this technique, we get the millionth decimal, 2, in 25 seconds and the billionth decimal, 8, in 19 hours. Note that we could further parallelize inside the individual sums to compute partial sums and then use Coq theorems to glue them together.

3 Algorithms to compute π based on arithmetic geometric means

In principle, all the mathematics that we had to describe formally in our study of arithmetic geometric means and the number π are available from the

mathematical litterature, essentially from the monograph by J. M. Borwein and P. B. Borwein [13] and the initial papers by R. Brent [14], E. Salamin [37]. However, we had difficulties using these sources as a reference, because they rely on an extensive mathematical culture from the reader. As a result, we were actually guided by a variety of sources on the world-wide web, including an exam for the selection of French high-school mathematical teachers [2]. It feels useful to repeat these mathematical facts in a first section, hoping that they are exposed at a sufficiently elementary level to be understood by a wider audience. However, some details may still be missing from this exposition and they can be recovered from the formal development itself.

This section describes two algorithms, but their mathematical justification has a lot in common. The first algorithm that we present came to us as the object of an exam for high-school teachers [2], but in reality this algorithm is neither the first one to have been designed by mathematicians, nor the most efficient of the two. However, it is interesting that it brings us good tools to help proving the second one, which is actually more traditional (that second algorithm dates from 1976 [14,37], and it is the one implemented in the `mpfr` library [21]) and more efficient (we shall see that it requires much less divisions).

In a second part of our study, we concentrate on the accumulation of errors during the computations and show that we can also prove bounds on this. This part of our study is more original, as it is almost never covered in the mathematical litterature, however it re-uses most of the results we exposed in a previous article [5].

3.1 Mathematical basics for arithmetic geometric means

Here we enumerate a large collection of steps that make it possible to go from the basic notion of arithmetic-geometric means to the computation of a value of π, together with estimates of the quality of approximations.

This is a long section, consisting of many simple facts, but some of the detailed computations are left untold. Explanations given between the formulas should be helpful for the reader to recover most of the steps. However, missing information can be found directly in the actual formal development [8].

The arithmetic-geometric process. As already explained in section 1, the arithmetic-geometric mean of two numbers a and b is obtained by defining sequences a_n and b_n such that $a_0 = a$, $b_0 = b$ and

$$a_{n+1} = \frac{a_n + b_n}{2} \qquad b_{n+1} = \sqrt{a_n b_n}$$

A few tests using high precision calculators show that the two sequences a_n and b_n converge rapidly to a common value $M(a, b)$, with the number of common digits doubling at each iteration. The sequence a_n provides over approximations and the sequence b_n under approximations. Here is an example

computation (for each line, we stopped printing values at the first differing digit between a_n and b_n).

	a	b
0	1	0.5
1	0.75	0.70...
2	0.7285...	0.7282...
3	0.72839552...	0.72839550...
4	0.7283955155234534...	0.7283955155234533...

The function $M(a, b)$ also benefits from a scalar multiplication property:

$$M(ka, kb) = kM(a, b) \qquad M(a, b) = aM(1, \frac{b}{a}) \tag{8}$$

For the sake of computing approximations of π, we will mostly be interested in the sequences a_n and b_n stemming from $a_0 = 1$ and $b_0 = \frac{1}{\sqrt{2}}$.

Elliptic integrals. We will be interested in *complete elliptic integrals of the first kind*, noted $K(k)$. The usual definition of these integrals has the following form

$$K(k) = \int_0^{\frac{\pi}{2}} \frac{\mathrm{d}\theta}{\sqrt{1 - k^2 \sin^2 \theta}} \tag{9}$$

But it can be proved that the following equality holds, when setting $a = 1$ and $b = \sqrt{1 - k^2}$, and using a change of variable (we only use the form $I(a, b)$):

$$K(k) = I(a, b) = \int_0^{+\infty} \frac{\mathrm{d}t}{\sqrt{(a^2 + t^2)(b^2 + t^2)}} \tag{10}$$

Note that the integrand in I is symmetric, so that $I(a, b)$ is also half of the integral with infinities as bounds. With the change of variables $s = \frac{1}{2}(x - \frac{ab}{x})$, then reasoning by induction and taking the limit, we also have the following equalities

$$I(a, b) = I(\frac{a + b}{2}, \sqrt{ab}) = I(a_n, b_n) = I(M(a, b), M(a, b)) = \frac{\pi}{2M(a, b)}. \tag{11}$$

Equivalence when $x \to 0$ and derivatives. Another interesting property for elliptic integrals of the first kind can be obtained by the variable change $u = \frac{x}{t}$ on the integral on the right-hand side of equation (10).

$$I(1, x) = 2 \int_0^{\sqrt{x}} \frac{\mathrm{d}t}{\sqrt{(1 + t^2)(x + t^2)}} \tag{12}$$

Studying this integral when x tends to 0 gives the equivalences for I and M:

$$I(1, x) \sim 2 \ln(\frac{1}{\sqrt{x}}) \quad M(1, x) \sim \frac{-\pi}{2 \ln x} \qquad \text{when } x \to 0^+. \tag{13}$$

For the rest of this section, we will assume that x is a value in the open interval $(0, 1)$ and that $a_0 = 1$ and $b_0 = x$. Coming back to the sequences a_n and b_n, the following property can be established.

$$M\left(a_{n+1}, \sqrt{a_{n+1}^2 - b_{n+1}^2}\right) = \frac{1}{2} M\left(a_n, \sqrt{a_n^2 - b_n^2}\right) \tag{14}$$

We can repeat n times and use the fact that $a_0^2 - b_0^2 = 1 - x^2$.

$$2^n M\left(a_n, \sqrt{a_n^2 - b_n^2}\right) = 2^n a_n M\left(1, \frac{\sqrt{a_n^2 - b_n^2}}{a_n}\right) = M\left(1, \sqrt{1 - x^2}\right) \tag{15}$$

Still under the assumption of $a_0 = 1$ and $b_0 = x$, we define k_n as follows:

$$k_n(x) = \frac{\ln\left(\frac{a_n}{\sqrt{a_n^2 - b_n^2}}\right)}{2^n} \tag{16}$$

Through separate calculation, involving Equation (13) and the definition of k, we establish the following properties.

$$\lim_{n\to\infty} k_n(x) = \frac{\pi}{2} \frac{M(1, x)}{M(1, \sqrt{1 - x^2})} \qquad k_n' = \frac{b_n^2}{x(1 - x^2)} \tag{17}$$

These derivatives converge uniformly to their limit. Moreover, the sequence of derivatives of a_n is growing and converges uniformly. This guarantees that $x \mapsto M(1, x)$ is also differentiable and and its derivative is the limit of the derivatives of a_n. We can then obtain the following two equations, the second is our main central formula.

$$\left(\frac{\pi}{2} \frac{M(1, x)}{M(1, \sqrt{1 - x^2})}\right)' = \frac{M(1, x)^2}{x(1 - x^2)} \qquad \pi = 2\sqrt{2} \frac{M(1, \frac{1}{\sqrt{2}})^3}{(M(1, x))'(\frac{1}{\sqrt{2}})} \tag{18}$$

We define the functions $y_n = \frac{a_n}{b_n}$ and $z_n = \frac{b_n'}{a_n'}$. These sequence satisfy

$$y_0 = \frac{1}{x} \qquad y_{n+1} = \frac{1 + y_n}{2\sqrt{y_n}} \qquad z_1 = \frac{1}{\sqrt{x}} \qquad z_{n+1} = \frac{1 + z_n y_n}{(1 + z_n)\sqrt{y_n}} \tag{19}$$

and the following important chain of comparisons.

$$y_{n+1} \leq z_{n+1} \leq \sqrt{y_n} \tag{20}$$

Computing with y_n and z_n (the Borwein algorithm). The first algorithm we will present, proposed by J. M. Borwein and P. B.Borwein, consists in approximating M using the sequences y_n and z_n. The value $M(1,x)^3$ is approximated using $a_n b_n^2$ and $(M(1,x))'$ using a'_n, all values being taken in $\frac{1}{\sqrt{2}}$.

From the definition, we can easily derive the following properties:

$$1 + y_n = 2\frac{a_{n+1}b_{n+1}^2}{a_n b_n^2} \qquad 1 + z_n = 2\frac{a'_{n+1}}{a'_n} \tag{21}$$

Repeating the products, we get the following definition of a sequence π_n and the proof of its limit:

$$\pi_0 = (2+\sqrt{2}) \qquad \pi_n = \pi_0 \prod_{i=1}^{n} \frac{1+y_i}{1+z_i} \qquad \lim_{n\to\infty} \pi_n = \pi \tag{22}$$

Convergence speed. For an arbitrary x in the open interval $(0,1)$, using a Taylor expansion of the function $y \mapsto \frac{1+y}{2\sqrt{y}}$ of order two, and then reasoning by induction, we get the following results:

$$y_{n+1}(x) - 1 \leq \frac{(y_n(x)-1)^2}{8} \qquad y_{n+1}(x) \leq 8\left(\frac{(y_1(x)-1)}{8}\right)^{2^n} \tag{23}$$

For $x = \frac{1}{\sqrt{2}}$, we obtain the following bound:

$$y_{n+1}(\frac{1}{\sqrt{2}}) - 1 \leq 8 \times 531^{-2^n} \tag{24}$$

Using the comparisons of line (20) and then reasoning by induction we obtain our final error estimate:

$$0 \leq \pi_{p+1} - \pi \leq \pi_{p+1}\left(y_{p+1}(\frac{1}{\sqrt{2}}) - 1\right) \leq 4\pi_0 531^{-2^p} \tag{25}$$

Computing one million decimals. The first element of the sequence π_n that is close to π with an error smaller than 10^{10^6} is obtained for n satisfying the following comparison.

$$n \geq \frac{\ln\left(\frac{10^6 \ln 10 - \ln(4\pi_0)}{\ln 531}\right)}{\ln 2} \sim 18.5 \tag{26}$$

For one million hexadecimals, n only needs to be larger than 18.75.

Computing with an infinite sum (the Brent-Salamin algorithm). The formula described in this section probably appears in Gauss' work and is repeated by King [31]. It was published and clarified for implementation on modern computers by Brent [14] and Salamin [37]. A good account of the historical aspects is given by Almkvist and Berndt [1]. Our presentation relies on a mathematical exposition given by Gourevitch [23].

In the variant proposed by Brent and Salamin, we compute the right-hand side of the main central formula by computing a_n^2 and the ratio $\frac{b'_n}{b_n}$. We first introduce a third function c_n.

$$c_n = \frac{1}{2}(a_{n-1} - b_{n-1}) \tag{27}$$

The derivative of function k_n can be expressed with c_n and after combination with equation 17, this gives a formula for the derivative of $\frac{a_n}{b_n}$ at $\frac{1}{\sqrt{2}}$.

$$\left(\frac{a_n}{b_n}\right)'\left(\frac{1}{\sqrt{2}}\right) = \frac{-2^{n+1}\sqrt{2}\,a_n c_n^2}{b_n} \tag{28}$$

The derivative of this ratio can be compared to the difference of the ratio of b'_n over b_n at two successive indices, which can be repeated n times.

$$\frac{b'_{n+1}}{b_{n+1}} - \frac{b'_n}{b_n} = \frac{b_n}{2a_n}\left(\frac{a_n}{b_n}\right)' \qquad \frac{b'_{n+1}}{b_{n+1}} = \frac{b'_1}{b_1} - \sqrt{2}\sum_{k=1}^{n-1} 2^k c_n^2 \tag{29}$$

We can then use equations (27) and (18), where $M^3(1, \frac{1}{\sqrt{2}})$ is the limit of $a_n^2 b_n$ to obtain the final definition and limit.

$$\pi'_n = \frac{4\,a_n^2}{1 - \sum_{k=1}^{n-1} 2^{k-1}(a_{k-1} - b_{k-1})^2} \qquad \pi = \lim_{n\to\infty} \pi'_n \tag{30}$$

Speed of convergence. We can link the Brent-Salamin algorithm with the Borwein algorithm in the following manner:

$$\pi'_n = 2\sqrt{2}\,\frac{a_n^2 b_n}{b'_n} = 2\sqrt{2}\,\frac{y_n}{z_n}\frac{a_n b_n^2}{a'_n} = \frac{y_n}{z_n}\pi_n \tag{31}$$

Combining bounds (20), (24), and (25) we obtain this first approximation.

$$|\pi'_{n+1} - \pi| \leq 68 \times 531^{-2^{n-1}} \tag{32}$$

This first approximation is too coarse, as it gives the impression that π'_{n+1} is needed when π_n is enough (the exponent of 2 in bound (32) is $n-1$ while it is n in bound (25)). We can make it better by noting that the difference between π and π_{n+2} is $O(531^{-2^n})$ and the difference between π_{n+2} and π_{n+1} is significantly smaller than $531^{-2^{n-1}}$, while not being $O(531^{-2^n})$.

$$|\pi'_{n+1} - \pi| \leq (132 + 384 \times 2^n) \times 531^{-2^n} \tag{33}$$

For one million decimals of π, we can still use $n = 19$.

Each algorithm computes n square roots to compute π_n or π'_n. However, the first one uses $3n$ division to obtain value π_n, while the second one only performs divisions by 2, which are less costly, and a single full division at the end of the computation to compute π'_n. In our experiments computing these algorithms inside Coq, the second one is twice as fast.

3.2 Formalization issues for arithmetic geometric means

In this section, we describe the parts of our development where we had to proceed differently from the mathematical exposition in section 3.1. Many difficulties arose from gaps in the existing libraries for real analysis.

The arithmetic geometric mean functions. For a given $a_0 = a$ and $b_0 = b$, the functions a_n and b_n actually are functions of a and b that are defined mutually recursively. Instead of a mutual recursion between two functions, we chose to simply describe a function `ag` that takes three arguments and returns a pair of two arguments. This can be written in the following manner:

```
Fixpoint ag (a b : R) (n : nat) :=
  match n with
    0 => (a, b)
  | S p => ag ((a + b) / 2) (sqrt (a * b)) p
  end.
```

This functions takes three arguments, two of which are real numbers, and the third one is a natural number. When the natural number is 0, then the result is the pair of the real numbers, thus expressing that $a_0 = a$ and $b_0 = b$. When the natural number is the successor of some p, then the two real number arguments are modified in accordance to the arithmetic-geometric mean process, and then the p-th argument of the sequence starting with these new values is computed.

As an abbreviation we also use the following definitions, for the special case when the first input is 1.

```
Definition a_ (n : nat) (x : R) := fst (ag 1 x n).

Definition b_ (n : nat) (x : R) := snd (ag 1 x n).
```

The function `ag_step` seems to perform the operation in a different order, but in fact we can really show that $a_{n+1} = \frac{a_n+b_n}{2}$ and $b_{n+1} = \sqrt{a_n b_n}$ as expected, thanks to a proof by induction on n. This is expressed with theorems of the following form:

```
Lemma a_step n x : a_ (S n) x = (a_ n x + b_ n x) / 2.

Lemma b_step n x : b_ (S n) x = sqrt (a_ n x * b_ n x).
```

Limits and filters. Cartan [15] proposed in 1937 a general notion that made it possible to develop notions of limits in a uniform way, whether they concern limits of continuous function of limits of sequences. This notion, known as *filters* is provided in formalized mathematics in Isabelle [30] and more recently in the Coquelicot library [11]. It is also present in a simplified form as *convergence nets* in Hol-Light [28].

Filters are not real numbers, but objects designed to represent ways to approach a limit. There are many kinds of filters, attached to a wide variety of types, but for our purposes we will mostly be interested in seven kinds of filters.

- `eventually` represents the limit towards ∞, but only for natural numbers,
- `locally` x represents a limit approaching a real number x from any side,
- `at_point` x represents a limit that is actually not a limit but an exact value: you approach x because you are bound to be exactly x,
- `at_right` x represents a limit approaching x from the right, that is, only taking values that are greater than x (and not x itself),
- `at_left` x represents a limit approaching x from the left,
- `Rbar_locally p_infty` describes a limit going to $+\infty$,
- `Rbar_locally m_infty` describes a limit going to $-\infty$.

There is a general notion called `filterlim` f F_1 F_2 to express that the value returned by f tends to a value described by the filter F_2 when its input is described by F_1. For instance, we constructed formal proofs for the following two theorems:

```
Lemma lim_atan_p_infty :
  filterlim atan (Rbar_locally p_infty) (at_left (PI / 2)).

Lemma lim_atan_m_infty :
  filterlim atan (Rbar_locally m_infty) (at_right (-PI / 2)).
```

In principle, filters make it possible to avoid the usual $\varepsilon - \delta$ proofs of topology and analysis, using faster techniques to relate input and output filters for continuous functions [30]. In practice, for precise proofs like the ones above (which use the `at_right` and `at_left` filters), we still need to revert to a traditional $\varepsilon - \delta$ framework.

Improper integrals. The Coq standard library of real numbers has been providing proper integrals for a long time, more precisely *Riemann integrals*. The Coquelicot library adds an incomplete treatement of improper integrals on top of this. For improper integrals the bounds are described as limits rather than as direct real numbers. For the needs of this experiment, we need to be able to cut improper integrals into pieces, perform variable changes, and compute the improper integral

$$\int_{-\infty}^{+\infty} \frac{\mathrm{d}t}{1+t^2} = \pi \tag{34}$$

The Coquelicot library provides two predicates to describe improper integrals, the first one has the form[1]

`is_Rint_gen` f B_1 B_2 v

The meaning of this predicate is "the improper integral of function f between bounds B_1 and B_2 converges and has value v". The second predicate is named `ex_Rint_gen` and it simply takes the same first three arguments as `is_Rint_gen`, to express that there exists a value v such that `is_Rint_gen` holds. The Coquelicot library does not provide a functional form, but there is a general tool to construct functions from relations where one argument is uniquely determined by the others, called `iota` in that library.

Concerning elliptic integrals, as a first step we need to express the convergence of the improper integral in equation (10). For this we need a general theorem of bounded convergence, which is described formally in our development, because it is not provided by the library. Informally, the statement is that the improper integral of a positive function is guaranteed to converge if that function is bounded above by another function that is known to converge. Here is the formal statement of this theorem:

```
Lemma ex_RInt_gen_bound (g : R -> R) (f : R -> R) F G
  {PF : ProperFilter F} {PG : ProperFilter G} :
  filter_Rlt F G ->
  ex_RInt_gen g F G ->
  filter_prod F G
    (fun p => (forall x, fst p < x < snd p -> 0 <= f x <= g x) /\
       ex_RInt f (fst p) (snd p)) ->
    ex_RInt_gen f F G.
```

This statement exhibits a concept that we needed to devise, the concept of comparison between filters on the real line, which we denote `filter_Rlt`. This concept will be described in further detail in a later section. Three other lines in this theorem statement deserve more explanations, the lines starting at `filter_prod`. These lines express that a property must ultimately be satisfied for pairs p of real numbers whose components tend simultaneously to the limits described by the filters `F` and `G`, which here also serve as bounds for two generalized Riemann integrals. This property is the conjunction of two facts, first for any argument between the pair of numbers, the function `f` is non-negative and less than or equal to `g` at that argument, second the function `f` is Riemann-integrable between the pair of numbers.

Using this theorem of bounded convergence, we can prove that the function

$$x \mapsto \frac{1}{\sqrt{(x^2+a^2)(x^2+b^2)}}$$

[1] the name can be decomposed in `is R` for Riemann, `Int` for Integral, and `gen` for generalized.

is integrable between $-\infty$ and $+\infty$ as soon as both a and b are positive, using the function

$$x \mapsto \frac{1}{m^2((\frac{x}{m})^2 + 1)}$$

as the bounding function, where $m = min(a, b)$, and then proving that this one is integrable by showing that its integral is related to the arctangent function.

Having proved the integrability, we then define a function that returns the following integral value:

$$\int_{-\infty}^{+\infty} \frac{\mathrm{d}x}{\sqrt{(x^2 + a^2)(x^2 + b^2)}}$$

The definition is done in the following two steps:

```
Definition ellf (a b : R) x :=
   /sqrt ((x ^ 2 + a ^ 2) * (x ^ 2 + b ^ 2)).

Definition ell (a b : R) :=
  iota (fun v => is_RInt_gen (ellf a b)
             (Rbar_locally m_infty) (Rbar_locally p_infty) v).
```

The value of `ell` a b is properly defined when a and b are positive. This is expressed with the following theorems, and will be guaranteed in all other theorems where `ell` occurs.

```
Lemma is_RInt_gen_ell a b : 0 < a -> 0 < b ->
  is_RInt_gen (ellf a b)
     (Rbar_locally m_infty) (Rbar_locally p_infty) (ell a b).

Lemma ell_unique a b v : 0 < a -> 0 < b ->
  is_RInt_gen (ellf a b)
     (Rbar_locally m_infty) (Rbar_locally p_infty) v ->
  v = ell a b.
```

An order on filters. On several occasions, we need to express that the bounds of improper integrals follow the natural order on the real line. However, these bounds may refer to no real point. For instance, there is no real number that corresponds to the limit 0+, but it is still clear that this limit represents a place on the real line which is smaller than 1 or $+\infty$. This kind of comparison is necessary in the statement of `ex_RInt_gen_bound`, as stated above, because the comparison between functions would be vacuously true when the bounds of the interval are interchanged.

We decided to introduce a new concept, written `filter_Rlt` F G to express that when x tends to F and y tends to G, we know that ultimately $x < y$. To be more precise about the definition of `filter_Rlt`, we need to know more about the nature of filters.

Filters simply are sets of sets. Every filter contains the complete set of elements of the type being considered, it is stable by intersection, and it is

stable by the operation of taking a superset. Moreover, when a filter does not contain the empty set, it is called a *proper filter*. For instance, the filter `Rbar_locally p_infty` contains all intervals of the form $(a, +\infty)$ and their supersets, the filter `locally x` contains all open balls centered in `x` and their supersets, and the filter `at_right x` contains the intersections of all members of `locally x` with the interval $(\mathtt{x}, +\infty)$.

With two filters F_1 and F_2 on types T_1 and T_2, it is possible to construct a product filter on $T_1 \times T_2$, which contains all cartesian products of a set in F_1 and a set in F_2 and their supersets. This corresponds to pairs of points which tend simultaneously towards the limits described by F_1 and F_2.

To define a comparison between filters on the real line, we state that F_1 is less than F_2 if there exists a middle point m, so that the product filter $F_1 \times F_2$ accepts the set of pairs v_1, v_2 such that $v_1 < m < v_2$. In other words, this means that as v_1 tends to F_1 and v_2 to F_2, it ultimately holds that $v_1 < m < v_2$. In yet other words, if there exists an m such that the filter F_1 contains $(-\infty, m)$ and F_2 contains $(m, +\infty)$, then F_1 is less than F_2. These are expressed by the following definition and the following theorem:

```
Definition filter_Rlt F1 F2 :=
  exists m, filter_prod F1 F2 (fun p => fst p < m < snd p).

Lemma filter_Rlt_witness m (F1 F2  : (R -> Prop) -> Prop) :
  F1 (Rgt m) -> F2 (Rlt m) ->  filter_Rlt F1 F2.
```

We proved a few comparisons between filters, for instance `at_right` x is smaller than `Rbar_locally p_infty` for any real x, `at_left` a is smaller than `at_right` b if $a \leq b$, but `at_right` c is only smaller than `at_left` d when $c < d$.

We can reproduce for improper integrals the results given by the Chasles relations for proper Riemann integrals. Here is an example of a Chasles relation: *if* f is integrable between a and c and $a \leq b \leq c$, then f is integrable between a and b and between b c, and the integrals satisfy the following relation:

$$\int_a^c f(x)\,\mathrm{d}x = \int_a^b f(x)\,\mathrm{d}x + \int_b^c f(x)\,\mathrm{d}x$$

This theorem is provided in the Coquelicot library for a, b, and c taken as real numbers. With the order of filters, we can simply re-formulate this theorem for a and c being arbitrary filters, and b being a real number between them. This is expressed as follows:

```
Lemma ex_RInt_gen_cut (a : R) (F G: (R -> Prop) -> Prop)
   {FF : ProperFilter F} {FG : ProperFilter G} (f : R -> R) :
   filter_Rlt F (at_point a) -> filter_Rlt (at_point a) G ->
   ex_RInt_gen f F G -> ex_RInt_gen f (at_point a) G.
```

We are still considering whether this theorem should be improved, using the filter `locally` a instead of `at_point` a for the intermediate integration bound.

The theorem `ex_RInt_gen_cut` is used three times, once to establish equation (11) and twice to establish equation (12) at page 12.

From improper to proper integrals. Through variable changes, improper integrals can be transformed into proper integrals and vice-versa. For instance, the change of variable leading to equation (11) actually leads to the correspondence.

$$\int_0^{+\infty} \frac{\mathrm{d}t}{\sqrt{(a^2+t^2)(b^2+t^2)}} = \frac{1}{2}\int_{-\infty}^{+\infty} \frac{\mathrm{d}s}{\sqrt{((\frac{a+b}{2})^2+s^2)(ab+s^2)}}$$

The lower bounds of the two integrals correspond to each other with respect to the variable change $s = \frac{1}{2}(t - \frac{ab}{t})$, but the first lower bound needs to be considered proper for later uses, while the lower bound for the second integral is necessarily improper. To make it possible to change from one to the other, we establish a theorem that makes it possible to transform a limit bound into a real one.

```
Lemma is_RInt_gen_at_right_at_point (f : R -> R) (a : R) F
  {FF : ProperFilter F v} :
  locally a (continuous f) -> is_RInt_gen f (at_right a) F v ->
  is_RInt_gen f (at_point a) F v.
```

This theorem contains an hypothesis stating that f should be well behaved around the real point being considered, the lower bound. In this case, we use an hypothesis of continuity *around* this point, but this hypothesis could probably be made weaker.

Limit equivalence. Equations (13.1) and (13.2) at page 12 rely on the concept of equivalent functions at a limit. For our development, we have not developed a separate concept for this, instead we expressed statements as the ratio between the equivalent functions having limit 1 when the input tends to the limit of interest. For instance equation (13.1) is expressed formally using the following lemma:

```
Lemma M1x_at_0 : filterlim (fun x => M 1 x / (- PI / (2 * ln x)))
                   (at_right 0) (locally 1).
```

In this theorem, the fact that x tends to 0 on the right is expressed by using the filter `(at_right 0)`.

We did not develop a general library of equivalence, but we still gave ourself a tool following the transitivity of this equivalence relation. This theorem is expressed in the following manner:

```
Lemma equiv_trans F {FF : Filter F} (f g h : R -> R) :
  F (fun x => g x <> 0) -> F (fun x => h x <> 0) ->
  filterlim (fun x => f x / g x) F (locally 1) ->
  filterlim (fun x => g x / h x) F (locally 1) ->
  filterlim (fun x => f x / h x) F (locally 1).
```

The hypotheses like `F (fun x => g x <> 0)` express that in the vicinity of the limit denoted by `F`, the function should be non-zero. The rest of the theorem expresses that if f is equivalent to g and g is equivalent to h, then f is equivalent to h. To perform this proof, we need to leave the realm of filters and fall back on the traditional $\varepsilon - \delta$ framework.

Uniform convergence and derivatives. During our experiments, we found that the concept of uniform convergence does not fit well in the framework of filters as provided by the Coquelicot library. The sensible approach would be to consider a notion of balls on the space of functions, where a function g is inside the ball centered in f if the value of $g(x)$ is never further from the value of $f(x)$ than the ball radius, for every x in the input type. One would then need to instantiate the general structures of topology provided by Coquelicot to this notion of ball, in particular the structures of `UniformSpace` and `NormedModule`. In practice, this does not provide all the tools we need, because we actually want to restrict the concept of uniform convergence to subsets of the whole type. In this case the structure of `UniformSpace` is still appropriate, but the concept of `NormedModule` is not, because two functions that differ outside the considered subset may have distance 0 when only considering their values inside the subset.

The alternative is provided by a treatment of uniform convergence that was developed in Coq's standard library of real numbers at the end of the 1990's, with a notion denoted `CVU` f g c r, where f is a sequence of functions from $\mathbb{R}$ to $\mathbb{R}$, g is a function from $\mathbb{R}$ to $\mathbb{R}$, c is a number in $\mathbb{R}$ and r is a positive real number. The meaning is that the sequence of function f converges uniformly towards g inside the ball centered in c of radius r. We needed a formal description of a theorem stating that when the derivatives f'_n of a convergent sequence of functions f_n tend uniformly to a limit function g', this function g' is the derivative of the limit of the sequence f_n.

There is already a similar theorem in Coq's standard library, with the following statement:

```
derivable_pt_lim_CVU :
forall fn fn' f g x c r,
Boule c r x ->
(forall y n, Boule c r y ->
                          derivable_pt_lim (fn n) y (fn' n y)) ->
(forall y, Boule c r y -> Un_cv (fun n : nat => fn n y) (f y)) ->
CVU fn' g c r ->
(forall y : R, Boule c r y -> continuity_pt g y) ->
derivable_pt_lim f x (g x)
```

However, this theorem is sometimes impractical to use, because it requires that we already know the limit derivative to be continuous, a condition that can actually be removed. For this reason, we developed a new formal proof for the theorem, with the following statement[2]

```
Lemma CVU_derivable :
forall f f' g g' c r,
 CVU f' g' c r ->
 (forall x, Boule c r x -> Un_cv (fun n => f n x) (g x)) ->
```

[2] It turns out that the theorem `derivable_pt_lim_CVU` was already introduced by a previous study on the implementation of π in the Coq standard library of real numbers [6].

```
(forall n x, Boule c r x ->
                    derivable_pt_lim (f n) x (f' n x)) ->
forall x, Boule c r x -> derivable_pt_lim g x (g' x).
```

In this theorem's statement, the third line expresses that the derivatives `f'` converge uniformly towards the function `g'`, the fourth line expresses that the functions `f` converge simply towards the function `g` inside the ball of center `c` and radius `r`, the fifth and sixth line express that the functions `f` are differentiable everywhere inside the ball and the derivative is `f'`, and the seventh line concludes that the function `g` is differentiable everywhere inside the ball and the derivative is `g'`. While most of the theorems we wrote are expressed using concepts from the Coquelicot library, this one is only expressed with concepts coming from Coq's standard library of real numbers, but all these concepts, apart from `CVU`, have a Coquelicot equivalent (and Coquelicot provides the foreign function interface): `Boule c r x` is equivalent to `Ball c r x` in Coquelicot, `Un_cv` f l is equivalent to `filterlim` f `Eventually (locally` l`)`, and `derivable_pt_lim` is equivalent to `is_derive`.

We used the theorem `CVU_derivable` twice in our development, once to establish that function $x \mapsto M(1, x)$ is differentiable everywhere in the open interval $(0, 1)$ and the sequence of derivatives of the a_n functions converges to its derivative, and once to establish that the derivatives of the k_n functions converge to $M^2(1, x)/(x(1 - x^2))$, as in equation (18).

Automatic proofs. In this development, we make an extensive use of divisions and square root. To reason about these functions, it is often necessary to show that the argument is non-zero (for division), or positive (for square root). There are very few automatic tools to establish this kind of results in general about real numbers, especially in our case, where we rely on a few transcendental functions. For linear arithmetic formulas, there exists a tool call `psatzl R` [9], that is very useful and robust in handling of conjunctions and its use of facts from the current context. Unfortunately, we have many expressions that are not linear. We decided to implement a semi-automatic tactic for the specific purpose of proving that numbers are positive, with the following ordered heuristics:

- Any positive number is non-zero,
- all exponentials are positive,
- π, 1, and 2 are positive,
- the power, inverse, square root of positive numbers is positive,
- the product of positive numbers is positive,
- the sum of an absolute value or a square and a positive number is positive,
- the sum of two positive numbers are positive,
- the minimum of two positive numbers is positive,
- a number recognized by the `psatzl R` tactic to be positive is positive.

This semi-automatic tactic can easily be implemented using Coq's tactic programming language `Ltac`. We named this tactic `lt0` and it is used extensively in our development.

Given a function like $x \mapsto 1/\sqrt{(x^2+a^2)(x^2+b^2)}$, the Coquelicot library provides automatic tools (mainly a tactic called `auto_derive`) to show that this function is differentiable under conditions that are explicitly computed. For this to work, the tool needs to rely on a database of facts concerning all functions involved. In this case, the database must of course contain facts about exponentiation, square roots, and the inverse function. As a result, the tactic `auto_derive` produces conditions, expressing that $(x^2+a^2)(x^2+b^2)$ must be positive and the whole square root expression must be non zero.

The tactic `auto_derive` is used more than 40 times in our development, mostly because there is no automatic tool to show the continuity of functions and we rely on a theorem that states that any differentiable function is continuous, so that we often prove derivability only to prove continuity.

When proving that the functions a_n and b_n are differentiable, we need to rely on a more elementary proof tool, called `auto_derive_fun`. When given a function to derive, which contains functions that are not known in the database, it builds an extra hypothesis, which says that the whole expression is differentiable as soon as the unknown functions are differentiable. This is especially useful in this case, because the proof that b_n is differentiable is done recursively, so that there is no pre-existing theorem stating that a_n and b_n are differentiable when studying the derivative of b_{n+1}. For instance, we can call the following tactic:

```
auto_derive_fun (fun y => sqrt (a_ n y * b_ n y)); intros D.
```

This creates a new hypothesis named `D` with the following statement:

```
D : forall x : R,
    ex_derive (fun x0 : R => a_ n x0) x /\
    ex_derive (fun x0 : R => b_ n x0) x /\
    0 < a_ n x * b_ n x /\ True ->
    is_derive (fun x0 : R => sqrt (a_ n x0 * b_ n x0)) x
      ((1 * Derive (fun x0 : R => a_ n x0) x * b_ n x +
        a_ n x * (1 * Derive (fun x0 : R => b_ n x0) x)) *
       / (2 * sqrt (a_ n x * b_ n x)))
```

Another place where automation provides valuable help is when we wish to find good approximations or bounds for values. The `interval` tactic [34] works on goals consisting of such comparisons and solves them right away, as long as it knows about all the functions involved. Here is an example of a comparison that is easily solved by this tactic:

```
(1 + ((1 + sqrt 2)/(2 * sqrt (sqrt 2))))
       / (1 + / sqrt (/ sqrt 2)) < 1
```

An example of expression where `interval` fails, is when the expressions being considered are far too large. In our case, we wish to prove that

$$4\pi_0 \frac{1}{531^{2^{19}}} \leq \frac{1}{10^{10^6+4}}$$

The numbers being considered are too close to 0 for `interval` to work.

The solution to this problem is to first use monotonicity properties of either the logarithm function (in the current version of our development) or the exponential function (in the first version), thus resorting to symbolic computation before finishing off with the `interval` tactic.

The `interval` tactic already knows about the π constant, so that it is quite artificial to combine our result from formula (25) and this tactic to obtain approximations of π but we can still make this experiment and establish that the member π_3 of the sequence is a good enough approximation to know all first 10 digits of π. Here is the statement:

```
Lemma first_computation :
   3141592653/10 ^ 9 < agmpi 3 /\
     agmpi 3 + 4 * agmpi 0 * Rpower 531 (- 2 ^ 2)
    < 3141592654/10 ^ 9.
```

We simply expand fully `agmpi`, simplify instances of y_n and z_n using the equations (21), and then ask the `interval` tactic to finish the comparisons. We need to instruct the tactic to use 40 bits of precision. This takes some time (about a second for each of the two comparisons), and we conjecture that the expansion of all functions leads to sub-expression duplication, leading also to duplication of work. When aiming for more distant decimals, we will need to apply another solution.

4 Computing large numbers of decimals

Theorem provers based on type theory have the advantage that they provide computation capabilities on inductive types. For instance, the Coq system provides a type of integers that supports comfortable computations for integers with size going up to 10^{100}. Here is an example computation, which feels instantaneous to the user.

```
Compute (2 ^331)%Z.
     = 1749800579826409539498001781694097092282535544714569949140
  616485127962399359500738578810541618443059 2
     : Z
```

By their very nature, real numbers cannot be provided as an inductive datatype in type theory. Thus the `Compute` command will not perform any computation for the similar expression concerning real numbers. The reason is that while some real numbers are defined like integers by applying simple finite operations on basic constants like 0 and 1, other are only obtained by applying a limiting process, which cannot be represented by a finite computation. Thus, it does not make sense to ask to compute an expression like $\sqrt{2}$ in the real numbers, because there is no way to provide a better representation of this number than its definition. On the other hand, what we usually mean by *computing* $\sqrt{2}$ is to provide a suitable approximation of this number. This is supported in the Coq system by the `interval` tactic, but only when we are in the process of constructing a proof, as in the following example:

```
Lemma anything : 12 / 10 < sqrt 2.
Proof.
interval_intro (sqrt 2).


1 subgoal

  H : 759250124 * / 536870912 <= sqrt 2 <= 759250125 * / 536870912
  ============================
   12 / 10 < sqrt 2
```

What we see in this dialog is that the system creates a new hypothesis (named `H`) that provides a new fact giving an approximation of $\sqrt{2}$. In this hypothesis, the common numerator appearing in both fractions is actually the number 2^{29}. Concerning notations, readers will have to know that Coq writes `/ a` for the inverse of `a`, so that `3 * / 2` is 3 times the inverse of 2. A human mathematician would normally write `3 / 2` and Coq also accepts this syntax.

One may argue that `759250124 * / 536870912` is not much better than `sqrt 2` to represent that number, and actually this ratio is not exact, but it can be used to help proving that $\sqrt{2}$ is larger or smaller than another number.

Direct computation on the integer datatype can also be used to approximate computations in real numbers. For instance, we can compute the same numerator for an approximation of $\sqrt{2}$ by computing the integer square root of $2 \times (2^{29})^2$.

```
Compute (Z.sqrt (2 * (2 ^ 29) ^ 2)).
     = 759250124%Z
     : Z
```

This approach of computing integer values for numerators of rational numbers with a fixed denominator is the one we are going to exploit to compute the first million digits of π, using three advantages provided by the Coq system:

1. The Coq system provides an implementation of *big integers*, which can withstand computations of the order of $10^{10^{12}}$.
2. The big integers library already contains an efficient implementation of integer square roots.
3. The Coq system provides a computation methodology where code is compiled into OCaml and then into binary format for fast computation.

4.1 A framework for high-precision computation

If we choose to represent every computation on real numbers by a computation on corresponding approximations of these numbers, we need to express how each operation will be performed and interpreted. We simply provide five values and functions that implement the elementary values of $\mathbb{R}$ and the elementary operations: multiplication, addition, division, the number 1, and the number 2.

We choose to represent the real number x by the integer $\lfloor mx \rfloor$ where m is a scaling factor that is mostly fixed for the whole computation. For readability, it is often practical to use a power of 10 as a scaling factor, but in this paper, we will also see that we can benefit from also using scaling factors that are powers of 2 or powers of 16. Actually, it is not even necessary that the scaling factor be any power of a small number, but it turns out that it is the most practical case.

Conversely, we shall note $[\![n]\!]$ the real value represented by the integer n. Simply, this number is $\frac{n}{m}$.

When m is the scaling factor, the real number 1 is represented by the integer m and the real number 2 is represented by the number $2 \times m$. So $[\![m]\!] = 1$, $[\![2m]\!] = 2$. So, we define the following two functions to describe the representations of 1 and 2 with respect to a given scaling factor, in Coq syntax where we use the name `magnifier` for the scaling factor.

```
Definition h1 (magnifier : bigZ) := magnifier.
Definition h2 magnifier := (2 * magnifier)%bigZ.
```

When multiplying two real numbers x and y, we need to multiply their representations and take care of the scaling. To understand how to handle the scaling, we should look at the following equality:

$$[\![n_1]\!][\![n_2]\!] = \frac{n_1}{m}\frac{n_2}{m}$$

To obtain the integer that will represent this result, we need to multiply the product of the represented numbers by m and then take the largest integer below. This is

$$\lfloor \frac{n_1 \times n_2}{m} \rfloor$$

The combination of the division operation and taking the largest integer below is performed by integer division. So we define our high-precision multiplication as follow.

```
Definition hmult (magnifier x y : bigZ) :=
  (x * y / magnifier)%bigZ.
```

For division and square root, we reason similarly.

For addition, nothing needs to be implemented, we can directly use integer computation. The scaling factor is transmitted naturally (and linearly from the operands to the result). Similarly, multiplication by an integer can be represented directly with integer multiplication, without having to first scale the integer.

Here are a few examples. To compute $\frac{1}{3}$ to a precision of 10^{-5}, we can run the following computation.

```
Compute let magnifier := (10 ^ 5)%bigZ in
  hdiv magnifier magnifier (3 * magnifier).
     = 33333%bigZ
     : BigZ.t_
```

The following illustrates how to compute $\sqrt{2}$ to the same precision.

```
Compute let magnifier := (10 ^ 5)%bigZ in
  hsqrt magnifier (2 * magnifier).
     = 141421%bigZ
     : BigZ.t_
```

In both examples, the real number of interest has the order of magnitude of 1 and is represented by a 5 or 6 digit integer. When we want to compute one million decimals of π we should handle integers whose decimal representation has approximately one million digits. Computation with this kind of numbers takes time. As an example, we propose a computation that handles the 1 million digit representation of $\sqrt{2}$ and avoids displaying this number (it only checks that the millionth decimal is odd).

```
Time Eval  native_compute in
   BigZ.odd (BigZ.sqrt (2 * 10 ^ (2 * 10 ^ 6))).
     = true
     : bool
Finished transaction in 91.278 secs (90.218u,0.617s) (successful)
```

This example also illustrates the use of a different evaluation strategy in the Coq system, called `native_compute`. This evaluation strategy relies on compiling the executed code in OCaml and then on relying on the most efficient variant of the OCaml compiler to produce a code that is executed and whose results are integrated in the memory of the Coq system [10]. This strategy relies on the OCaml compiler and the operating system linker in ways that are more demanding than traditional uses of Coq. Still it is the same compiler that is being used as the one used to compile the Coq system, so that the trusted base is not changed drastically in this new approach.

When it comes to time constraints, all scaling factors are not as efficient. In conventional computer arithmetics, it is well-known that multiplications by powers of 2 are less costly, because they can simply be implemented by shifts on the binary representation of numbers. This property is also true for Coq's implementation of big integers. If we compare the computation of $\sqrt{\sqrt{2}}$ when the scaling factor is 10^{10^6} or $2^{3321929}$, we get a performance ratio of 1.5, the latter setting is faster even though the scaling factor and the intermediate values are slightly larger.

It is also interesting to understand how to stage computations, so that we avoid performing the same computation twice. For this problem, we have to be careful, because values that are precomputed don't have the same size as their original description, and this may not be supported by the `native_compute` chain of evaluation. Indeed, the following experiment fails.

```
Require Import BigZ.

Definition mag := Eval native_compute in (10 ^ (10 ^ 6))%bigZ.

Time Definition z1 := Eval native_compute in
```

```
   let v := mag in (BigZ.sqrt (v * BigZ.sqrt (v * v * 2)))%bigZ.
```

This examples makes Coq fail, because the definition of `mag` with the pragma `Eval native_compute in` makes that the value 10^{10^6} is precomputed, thus creating a huge object of the Gallina language, which is then passed as a program for the OCaml compiler to compile when constructing `z1`. The compiler fails because the input program is too large.

On the other hand, the following computation succeeds:

```
Eval native_compute in
   let v := (10 ^ (10 ^ 6))%bigZ in
   (BigZ.sqrt (v * BigZ.sqrt (v * v * 2))).
```

4.2 The full approximating algorithm

Using all elementary operations described in the previous section, we can describe the recursive algorithm to compute approximations of π_n in the following manner.

```
Fixpoint hpi_rec (magnifier : bigZ)
  (n : nat) (s2 y z prod : bigZ) {struct n} : bigZ :=
  match n with
  | 0%nat =>
      hmult magnifier (h2 magnifier + s2) prod
  | S p =>
      let sy := hsqrt magnifier y in
      let ny := hdiv magnifier (h1 magnifier + y) (2 * sy) in
      let nz :=
        hdiv magnifier (h1 magnifier + hmult magnifier z y)
          (hmult magnifier (h1 magnifier + z) sy) in
      hpi_rec magnifier p s2 ny nz
        (hmult magnifier prod
           (hdiv magnifier (h1 magnifier + ny)
                           (h1 magnifier + nz)))
  end.
```

This function takes as input the scaling factor `magnifier`, a number of iteration `n`, the integer `s2` representing $\sqrt{2}$, the integer `y` representing y_p for some natural number p larger than 0, the integer `z` representing z_p, and the integer `prod` representing the value

$$\prod_{i=1}^{p} \frac{1 + y_i(\frac{1}{\sqrt{2}})}{1 + z_i(\frac{1}{\sqrt{2}})}$$

It computes an integer approximating $\pi_{n+p} \times$ `magnifier`, but not exactly this number. The number `s2` is passed as an argument to make sure it is not computed twice, because it is already needed to compute the initial values of `y`, `z`, and `prod`. This recursive function is wrapped in the following functions.

```
Definition hs2 (magnifier : bigZ) :=
  hsqrt magnifier (h2 magnifier).

Definition hsyz (magnifier : bigZ) :=
  let hs2 := hs2 magnifier in
  let hss2 := hsqrt magnifier hs2 in
  (hs2, (hdiv magnifier (h1 magnifier + hs2) (2 * hss2)), hss2).

Definition hpi (magnifier : bigZ) (n : nat) :=
match n with
| 0%nat =>
    (h2 magnifier + (hs2 magnifier))%bigZ
| S p =>
    let '(s2, y1 , z1) := hsyz magnifier in
    hpi_rec magnifier p s2 y1 z1
      (hdiv magnifier (h1 magnifier + y1)
         (h1 magnifier + z1))
end.
```

We can use this function `hpi` to compute approximations of π at a variety of precisions. Here is a collection of trials performed on a powerful machine.

scale(iterations)	$10^{10^4}(14)$	$2^{33220}(14)$	$10^{10^5}(17)$	$2^{332193}(17)$
time	9s	4s	5m30s	2m30s

This table illustrates the advantage there is to compute with a scaling factor that is a power of 2. Each column where the scaling factor is a power of 2 gives an approximation that is slightly more precise than the column to its left, at a fraction of the cost in time. Even if our objective is to obtain *decimals* of π, it should be efficient to first perform the computations of all the iterations with a magnifier that is a power of 2, only to change the scaling factor at the end of the computation, this is the solution we choose eventually.

There remains a question about how much precision is lost when so many computations are performed with elementary operations that each provide only approximations of the mathematical operation. Experimental evidence shows that when computing 17 iterations with a magnifier of 10^{10^5} the last two digits are wrong. The next section shows how we prove bounds on the accumulated error in the concrete computation.

5 Proofs about approximate computations

When proving facts about approximate computations, we want to abstract away from the fact that the computations are performed with a datatype that provides fast computation with big integers. What really matters is that we approximate each operation on real numbers by another operation on real numbers and we have a clear description of how the approximation works. In the next section, we describe the abstract setting and the proofs performed

in this setting. In a later section, we show how this abstract setting is related to the concrete setting of computing with integers and with the particular datatype of big integers.

5.1 Abstract reasoning on approximate computations

In the case of fixed precision computation as we described in the previous section, we know that all operations are approximated from below by a value which is no further than a fixed allowance e. This does not guarantee that all values are approximated from below, because one of the approximated operations is division, and dividing by an approximation from below may yield an approximation from above.

For this reason, most of our formal proofs about approximations are performed in a section where we assume the existence of a collection of functions and their properties.

The header of our working section has the following content.

```
Variables (e : R) (r_div : R -> R -> R) (r_sqrt : R -> R)
          (r_mult : R -> R -> R).

Hypothesis ce : 0 < e < /1000.

Hypothesis r_mult_spec :
  forall x y, 0 <= x -> 0 <= y ->
   x * y - e < r_mult x y <= x * y.
```

In this header, we introduce a constant `e`, which is used to bound the error made in each elementary operation, we assume that `e` is positive and suitably small, and then we describe how each rounded operation behaves with respect to the mathematical operation it is supposed to represent. For multiplication, the hypothesis named `r_mult_spec` describes that the inputs are expected to be positive numbers, and that the result of `r_mul x y` is smaller than or equal to the product, but the difference is smaller than `e` in absolute value. We have similar specification hypotheses for the rounded division `r_div` and the rounded square root `r_sqrt`. We then use these rounded operations to describe the computations performed in the algorithm.

We can now study how the computation of the various sequences of the algorithm are rounded, and how errors accumulate. Considering the sequence y_n, the computation at each step is represented by the following expression.

```
r_div (1 + y) (2 * (r_sqrt y))
```

In this expression, we have to assume that `y` comes from a previous computation, and for this reason it is tainted with some error `h`. The question we wish to address has the following form: *if we know that y_n is tainted with an error h that is smaller that a given allowance e', can we show that y_{n+1} is tainted with an error that is smaller than $f(e')$ for some well-behaved function f? How much bigger than e must e' be?*

We were able to answer two questions:

- if the accumulated error on computing y_n is smaller than e', then the accumulated error on computing y_{n+1} is also smaller than e' (so for the sequence y_n, the function f is the identity function),
- the allowance e' needs to be at least $2e$ (and not more).

This is quite surprising. Errors don't really accumulate for this sequence.

In retrospect, there are good reasons for this. Rounding errors in the division operation make the result go down, but rounding errors in the square root make the result go up. On the other hand, the input value y_n may be tainted by an error h, but this error is only multiplied by the derivative of the function

$$y \mapsto \frac{1+y}{2\sqrt{y}}$$

It happens that this derivative never exceeds $\frac{1}{14}$ in the region of interest.

As an illustration, let's assume $y_n = 1.100$, we want to compute y_{n+1}, and we are working with three digits of precision. The value of $\sqrt{1.1}$ is $1.04880\ldots$ but it is rounded down to 1.048. $2\sqrt{1.1}$ is $1.09761\ldots$ but the rounded computation give 2.096, y_{n+1} is 1.00113. In our computation, we actually compute $(1 + 1.1)/2.096) = 1.00190$. This is an over approximation of y_{n+1}, but this is rounded down to 1.001: the last rounding down compensates the over-approximation introduced when dividing by the previously rounded down square root. If our input representation of y_n is an approximation, for example we compute with 1.098 or 1.102, we still obtain 1.001.

In the end, the lemma we are able to prove has the following statement.

```
Lemma y_error e' y h :
  e' < /10 -> e <= e' / 2 -> 1 <= y <= 71/50 -> Rabs h < e' ->
  let y1 := (1 + y)/(2 * sqrt y) in
  y1 - e' < r_div (1 + (y + h)) (2 * (r_sqrt (y + h))) < y1 + e'.
```

The proof is organized in four parts, where the first part consists in replacing the operations with rounding by expressions where an explicit error ratio is displayed. We basically construct a value `e1`, taken in the interval $[-\frac{1}{2}, 0]$, so that the following equality holds.

```
r_sqrt (y + h) = sqrt (y + h) + e1 *  e'
```

We prefer to define `e1` as a ratio between constant bounds, rather than a value in an interval whose bounds are expressed in `e'`, because the automatic tactic `interval` handles values between numeric constants better. We do the same for the division, introducing a ratio `e2`.

The second part of the proof consists in showing that the propagated error from previous computations has limited impact on the final error. This is stated as follows.

```
set (y2 := (1 + (y + h)) / (2 * sqrt (y + h))).
assert (propagated_error : Rabs (y2 - y1) < e' / 14).
```

This step is proved by applying the mean value theorem, using the derivative of the function $y \mapsto \frac{1+y}{2\sqrt{y}}$, which was already computed during the proof of convergence of the y_n sequence. The `interval` tactic is practical here to show the absolute value of the derivative of that function at any point between `y` and `y + h` is below $\frac{1}{14}$. The mean value theorem makes it possible to factor out the input error in the comparisons, so that we eventually obtain a comparison of an expression with a constant, which we resolve using the `interval` tactic.

The other two parts of the proof are concerned with providing a bound for the impact of the rounding errors introduced by the current computation. Each part is concerned with one direction, and in each case only one of the two possible rounding errors need to be considered.

The proof for the lemma `y_error` is quite long (just under 100 lines), but this is only a preliminary step for the proof of lemma `z_error`, which shows that the errors accumulated when computing the z_n sequence can also be bounded in a constant fashion. The statement of this lemma has the following shape.

```
Lemma z_error e' y z h h' :
  e' < /50 -> e <= e' / 4 -> 1 < y < 51/50 -> 1 < z < 6/5 ->
  Rabs h < e' -> Rabs h' < e' ->
  let v := (1 + z * y)/((1 + z) * sqrt y) in
  v - e' < r_div (1 + r_mult (z + h') (y + h))
           (r_mult (1 + (z + h')) (r_sqrt (y + h))) < v + e'.
```

In this statement, the fragment

```
   r_div (1 + r_mult (z + h') (y + h))
       (r_mult (1 + (z + h') (r_sqrt (y + h)))
```

represents the computed expression with rounding operations, using inputs that are tainted by errors `h` and `h'`, while the fragment

```
(1 + z * y) /((1 + z) * sqrt y)
```

represents the ratio $\frac{1+zy}{(1+z)\sqrt{y}}$.

This proof is more complex. In this case, we are also able to show that errors do not grow as we compute more elements of the sequence: they stay stable at about 4 times the elementary rounding error introduced by each rounding operation. The proof of this lemma is around 170 lines long.

The next step in the computation is to compute the product of ratios $\prod \frac{1+y}{1+z}$. For each ratio, we establish a bound on the error as expressed by the following lemma.

```
Lemma quotient_error : forall e' y z h h', e' < / 40 ->
  Rabs h < e' / 2 -> Rabs h' < e' -> e <= e' / 4 ->
  1 < y < 51 / 50 -> 1 < z < 6 / 5 ->
  Rabs (r_div (1 + (y + h)) (1 + (z + h')) -
              (1 + y)/(1 + z)) <  13 / 10 * e'.
```

The difference between the second hypothesis (on `Rabs h`) and the third hypothesis `Rabs h'` handles the fact that we don't have as precise a bound on error for the computation of y_n and for z_n. The result is that the error on the ratio is bounded at a value just above 5 times the elementary error `e`.

It remains to prove a bound on the error introduced when computing the iterated product. This is done by induction on the number of iterations. The following lemma is used as the induction step: when `p` represents the product of k terms and `v` represents one of the ratios, the product of `p` and `v` with accumulated errors, adding the error for the rounded multiplication increases by $\frac{23}{20}$ the error on the ratio.

```
Lemma product_error_step :
  forall p v e1 e2 h h', 0 <= e1 <= /100 -> 0 <= e2 <= /100 ->
    e < /5 * e2 -> /2 < p < 921/1000 ->
    /2 < v <= 1 -> Rabs h < e1 -> Rabs h' < e2 ->
    Rabs (r_mult (p + h) (v + h') - p * v) < e1 + 23/20 * e2.
```

At this point we write functions `rpi_rec` and `rpi` so that they mirror exactly the functions `hpi_rec` and `hpi`. The main difference is that `rpi_rec` manipulates real numbers while `hpi_rec` manipulates integers. Aside from this, `rpi_rec` performs a multiplication using `r_mult` wherever `hpi_rec` performs a multiplication using `hmult`.

We can now combine all results about the sub-expressions, scale all errors with respect to the elementary error, and obtain a bound on accumulated errors in `rpi_rec`, as expressed in the following lemma.

```
Lemma rpi_rec_correct (p n : nat) y z prod :
    (1 <= p)%nat -> 4 * (3/2) * (p + n) * e < /100 ->
    Rabs (y - y_ p (/sqrt 2)) < 2 * e ->
    Rabs (z - z_ p (/sqrt 2)) < 4 * e ->
    Rabs (prod - pr p) < 4 * (3/2) * p * e ->
    Rabs (rpi_rec n y z prod - agmpi (p + n)) <
      (2 + sqrt 2) * 4 * (3/2) * (p + n) * e + 2 * e.
```

Note that this statement guarantees a bound on errors only if the magnitude of the error `e` is small enough when compared with the inverse of the number of iterations `p + n`. In practice, this is not a constraint because we tend to make the error magnitude vanish twice exponentially.

In the end, we have to check the approximations for the initial values given as argument to `rpi_rec`. This yields a satisfying rounding error lemma.

```
Lemma rpi_correct : forall n, (1 <= n)%nat -> 6 * n * e < /100 ->
  Rabs (rpi n - agmpi n) < (21 * n + 2) * e.
```

In other words, we can guarantee that π_n is computed with an error that grows proportionally to $21n + 2$.

A similar study for the *Brent-Salamin* algorithm yields the following error estimate:

```
Lemma rsalamin_correct (n : nat) :
 0 <= e <= / 10 ^ (n + 6) / 3 ^ (n + 1) ->
  Rabs (rsalamin n - salamin_formula (n + 1)) <=
  (160 * (3 / 2) ^ (n + 1) + 80 * 3 ^ (n + 1) + 100) * e.
```

This error grows exponentially with respect to n, which means that the number of needed extra digits to ensure a given distant decimal is still linear in n. When computing the number of required extra digits for 1 million, we obtain 12 (because n is 19).

5.2 From abstract rounding to integer computations

In our concrete setting, we don't have the functions `r_mult`, `r_div`, and `r_sqrt`, but functions `hmult`, `hdiv` and `hsqrt`. The type on which these functions operate is `bigZ`, a type that is designed to make large computations possible inside the Coq system, but that is otherwise not suited to perform intensive proofs. To establish the connection with our proofs of rounded operations, we build a bridge that relies on the better supported type `Z`.

The standard library of reals already provides function `INR` and `IZR` to inject natural numbers and integers, respectively, into the type of real numbers. These functions are useful to us, but they must be improved to include the scaling process.

We also define functions `hR : Z -> R` and `Rh : R -> Z` mapping an integer (respectively a real number) to its representation (respectively to the integer that represents its rounding by default). All these functions are defined in the context of a Coq section where we assume the existence of a scaling factor named `magnifier` (an integer), and that this scaling factor is larger than 1000, which corresponds to assuming that we perform computations with at least 3 digits of precision.

Coming from the type of integers, we can now redefine the functions `hmult`, `hdiv`, and `hsqrt` as in section 4.1, but with the type `Z` for inputs and outputs.

```
Definition hR (v : Z) : R := (IZR v /IZR magnifier)%R.

Definition RbZ (v : R) : Z := floor v.

Definition Rh (v : R) : Z := RbZ( v * IZR magnifier).
```

The abstract functions `r_mult`, `r_div` and `r_sqrt` are then defined by rounding and injecting the result back into the type of real numbers.

```
Definition r_mult (x y : R) : R := hR (Rh (x * y)).
```

The main rounding property can be proved once for all three rounded operations, since it is solely a property of the `hR` and `Rh` function.

```
Lemma hR_Rh (v : R) : v - /IZR magnifier < hR (Rh v) <= v.
```

The link to the concrete computing functions is established by the following kind of lemma, the form of which is close to a morphism lemma.

```
Lemma hmult_spec :
  forall x y : Z, (0 <= x -> 0 <= y ->
   hR (hmult x y) = r_mult (hR x) (hR y))%Z.
```

The hypotheses `r_mult_spec`, `r_div_spec`, and `r_sqrt_spec`, which are necessary for the abstract reasoning in section 5.1, are then easily obtained by composing a lemma of the form `hmult_spec` with the lemma `hR_Rh`.

The complement of the lemma `hR_Rh` is another lemma which expresses that `Rh` is a left inverse to `hR`. This lemma is instrumental when showing the correspondence between concrete and abstract algorithms.

We now have two views of the algorithm: the algorithm `hpi` as described in section 4.1 and the algorithm `rpi` where the functions `hmult`, `hdiv`, `hsqrt` have been replaced by `r_mult`, `r_div`, `r_sqrt` respectively. We wish to show that these algorithms actually describe the same computation. A new difficulty arises because we need to show that all operations receive and produce non-negative numbers, because these conditions are required by lemmas like `hmult_spec`. This is not as simple as it seems because the result of `hmult 0 0` is only guaranteed to be larger than `- e` by the initial specification. The implementation actually satisfies a stronger property.

In the end the correspondence lemma has the following form.

```
Lemma hpi_rpi_rec n p y z prod:
    (1 <= p)%nat ->
    4 * (3/2) * INR (p + n) * /IZR magnifier < /100 ->
    Rabs (hR y - y_ p (/sqrt 2)) < 2 * /IZR magnifier ->
    Rabs (hR z - z_ p (/sqrt 2)) < 4 * /IZR magnifier ->
    Rabs (hR prod - pr p) < 4 * (3/2) * INR p * /IZR magnifier ->
    hR (hpi_rec n y z prod) =
    rpi_rec r_div r_sqrt r_mult n (hR y) (hR z) (hR prod).
```

The interesting part of this lemma is the equality stated on the last two lines. The previous lines only state information about the size of the inputs, to help make sure that the intermediate computations never feed a negative number to the operations. This constraint of non-negative operands makes the proof of correspondence tedious, but quite regular. This proof ends up being 120 lines long.[3]

A similar proof is constructed for the main encapsulating function, so that we obtain a lemma of the following shape.

```
Lemma hpi_rpi (n : nat) :
  6 * INR n * /IZR magnifier < / 100 ->
  hR (hpi n) = rpi r_div r_sqrt r_mult n.
```

[3] In retrospect, it might have been useful to add hypotheses that returned values by all functions were positive, as long as the inputs were.

```
Lemma integer_pi :
  forall n, (1 <= n)%nat ->
  600 * INR (n + 1) < IZR magnifier < Rpower 531 (2 ^ n)/ 14 ->
  Rabs (hR (hpi (n + 1)) - PI)
     < (21 * INR (n + 1) + 3) /IZR magnifier.
```

In the end, we obtain a description of the algorithm based on integers, which can be applied to any number of iterations and any suitable scaling factor. This algorithm can already be used to compute approximations of π inside Coq, but it will not return answers in reasonable time for precisions that go beyond a thousand digits (less than a second for a 7 iterations at 100 digits, 12 seconds for 9 iterations at 500 digits, a minute for 10 iterations at 1000 digits).

Concerning the magnitude of the accumulated error, for one million digits the number of iterations is 20, and the error is guaranteed to be smaller than 423.

Changing the scaling factor. Although we are culturally attracted by the fractional representation of π in decimal form, it is more efficient to perform most of the costly computations using a scaling factor that is a power of 2. For any two scaling factors m_1 and m_2, let us assume that v_1 and v_2 are linked by the equation

$$v_2 = \left\lfloor \frac{v_1 \times m_2}{m_1} \right\rfloor .$$

If v_1 is the representation of a constant a for the scaling factor m_1, then v_2 is a reasonably good approximation of a for the scaling factor m_2. This suggests that we could perform all operations with a scaling factor m_1 that is a power of 2 and then post-process the result to obtain a representation for the scaling factor m_2. Of course, one more multiplication and one more division need to be performed and a little precision is lost in the process, but the gain in computation time is worth it.

The validity of this change in scaling factor is expressed by the following lemma.

```
Lemma change_magnifier : forall m1 m2 x, (0 < m2)%Z ->
  (m2 < m1)%Z ->
  hR m1 x - /IZR m2 < hR m2 (x * m2/m1) <= hR m1 x.
```

This lemma expresses that the added error for this operation is only one time the inverse of the new scaling factor.

In our case, we use this lemma with `m1` $= 2^{3321942}$ and `m2` $= 10^{10^6+4}$ for instance.

Guaranteeing a fixed number of digits. When we want to compute a number N of digits, we don't know in advance whether the digits at position $N+1$, $N+2$, ... describe a small number or a large number. If this number is too

small or too large we are unable to guarantee the value of the digit at position N.

Let's illustrate this problem on a small example. Let's assume we want to compute the integral part of a and we have an approximated value b which is guaranteed to be within $1/4$ of a. Moreover, when computing b with a precision of 2 digits, we know that our computation process may introduce errors of two *units in the last place*. This means that we actually compute a value c whose distance to b is guaranteed to be smaller than 0.02. At the time we discover the result of computing c three cases may occur.

1. if the fractional part of c is smaller than 0.27, the integral part of a may be smaller than the integral part of c. For instance, we may have $c = 3.26$, $b = 3.245$, and $a = 2.995$
2. if the fractional part of c is larger than or equal to 0.73, the integral part of a may be larger than the integral part of c. For instance, we may have $c = 2.74$, $b = 2.755$, and $a = 3.005$.
3. if the fractional part of c is larger than or equal to 0.27 or smaller than 0.73, then we now that a, b, and c all share the same integral part.

When considering distant decimals, the same problem is transposed through multiplication by a large power of 10.

After putting together the error coming from the difference $\pi_n - \pi$, the accumulated rounding errors, and the error coming from the change of scaling factor, this means we need to verify that the last four digits are either larger than 0427 or smaller than 9573. This verification is made in the following definitions, which return a boolean value and a large integer. The meaning of the two values is expressed by the attached lemma.

```
Definition million_digit_pi : bool * Z :=
  let magnifier := (2 ^ 3321942)%Z in
  let n := hpi magnifier 20 in
    let n' := (n * 10 ^ (10 ^ 6 + 4) / 2 ^ 3321942)%Z in
    let (q, r) := Z.div_eucl n' (10 ^ 4) in
    ((427 <? r)%Z && (r <? 9573)%Z, q).

Lemma pi_osix :
  fst million_digit_pi = true ->
    hR (10 ^ (10 ^ 6)) (snd million_digit_pi) < PI <
    hR (10 ^ (10 ^ 6)) (snd million_digit_pi) +
    Rpower 10 (-(Rpower 10 6)).
```

Proving the big number computations. The lemma `million_digit_pi` only states the correctness of computations for computations in the type `Z`, but this computation is unpractical to perform. The last step is to obtain the same proof for computations on the type `bigZ`. The library `BigZ` provides both this type and a coercion function noted `[ · ]` so that when `x` is a big integer of type `bigZ`, `[x]` is the corresponding integer of type `Z`.

In what follows, the functions `rounding_big.hmult`, et cetera operate on numbers of type `BigZ`, while the functions `hmult` operate on plain integers. We have the following morphism lemmas:

```
Lemma hmult_morph p x y:
  [rounding_big.hmult p x y] = hmult [p] [x] [y].
Proof.
unfold hmult, rounding_big.hmult.
rewrite BigZ.spec_div, BigZ.spec_mul; reflexivity.
Qed.

Lemma hdiv_morph p x y:
  [rounding_big.hdiv p x y] = hdiv [p] [x] [y].
Proof.
unfold hdiv, rounding_big.hdiv.
rewrite BigZ.spec_div, BigZ.spec_mul; reflexivity.
Qed.
```

Using these lemmas, it is fairly routine to prove the correspondence between the algorithms instantiated on both types.

```
Lemma hpi_rec_morph :
 forall s p n v1 v2 v3,
   [s] = hsqrt [p] (h2 [p]) ->
   [rounding_big.hpi_rec p n s v1 v2 v3] =
   hpi_rec [p] n [s] [v1] [v2] [v3].

Lemma hpi_morph : forall p n,
   [rounding_big.hpi p n]%bigZ = hpi [p]%bigZ n.
```

In the end, we have a theorem that expresses the correctness of the computations made with big numbers, with the following statement.

```
Lemma big_pi_osix :
  fst rounding_big.million_digit_pi = true ->
  (IZR [snd rounding_big.million_digit_pi] *
         Rpower 10 (-(Rpower 10 6)) <
    PI
   <
    IZR [snd rounding_big.million_digit_pi]
      * Rpower 10 (-(Rpower 10 6))
     + Rpower 10 (-(Rpower 10 6)))%R.
```

This statement expresses that the computation returns a boolean value and a large integer. When this boolean value is `true`, then the large integer is the largest integer n so that

$$\frac{n}{10^{10^6}} < \pi.$$

The computation of this value takes approximately 2 hours on a powerful machine. We also implemented similar functions to compute approximations of π using the Brent-Salamin algorithm, and experiments showed the computation is twice as fast.

6 Related work

Computing approximations of π is a task that is necessary for many projects of formally verified mathematics, but precision beyond tens of digits are practically never required. To our knowledge, this work is the only one addressing explicitly the challenge of computing decimals at position beyond one thousand. Most developments rely on Machin-like formulas to give a computationally relevant definition of π. The paper [6] already provides an overview of methods used to compute π in a variety of provers. In Hol-Light [26], an approximation to the precision of 2^{-32} is obtained by approximating $\frac{\pi}{6}$ using the intermediate value theorem and a Taylor expansion of the sine function, and the library also provides a description of a variety of Machin-like formulas. In Isabelle/HOL [35], one of the Machin-like formulas is provided directly in the basic theory of transcendental functions. Computation of arbitrary mathematical formulas, in the spirit of what is done with the `interval` tactic, is described in work by Hölzl [29].

The HOL Light library contains a formalization of the BBP formula [27]. Our contribution is to link the formalization of the formula with the actual algorithm that computes the digit.

In the Coq system, real numbers can also be approached constructively as in the C-CoRN library [17]. This was used as the basis for a library providing fairly efficient computation of mathematical functions within the theorem prover [36,32]. Using an advanced Machin-like formula they are capable to compute numbers like $\sqrt{\pi}$ at a precision of 500 digits in about 6 seconds (to be compared with less than a second in our case, but our development is not as versatile as theirs).

The formalized proof of the Kepler conjecture, under the supervision of T. Hales [25] also required computing many inequalities between mathematical formulas involving transcendental functions, a task covered more specifically by Solovyev and Hales [39], but none of these computations involved precisions in the ranges that we have been studying here.

7 Conclusion

What we guarantee with our lemmas is that the integer we produce satisfies a property with respect to π and a large power of the base, which is 16 in the case of the the BBP algorithm, and may be any integer in the case of the algebraic-geometric mean algorithms. We do not guarantee that the string produced by the Coq system when printing this large number is correct, but

experimental evidence shows that that part of the Coq system (printing large numbers) is correct. That computations can proceed to the end is a nice surprise, because it would be understandable that some parts of the theorem prover have limitations that preclude heavy computing (as is the case when performing computations with natural numbers, which are notoriously naive in their implementation and their space and time complexity). It would be an interesting project to construct a formally verified integer to string converter, but this project is probably not as challenging as what has been presented in this article.

The organisation of proofs follows principles that were advocated by Cohen, Dénès, Mörtberg, and Siles [18,16], where the algorithm is first studied in a mathematical setting using mathematical objects (in this case real numbers) before being embodied in a more efficient implementation using different datatypes. The concrete implementation is then viewed as a refinement of the first algorithm. This approach makes sure that we take advantage of the most comfortable mathematical libraries when performing the most difficult proofs. The refinement approach was used twice: first to establish the correspondence between computations on real numbers and the computations on integers, and second to establish the correspondence between integers and big integers. The first stage does not fit exactly the framework advocated by Cohen and co-authors, because the computations are only approximated and we need to quantify the quality of the approximation. On the other hand, the second stage corresponds quite precisely to what they advocate, and it was a source of great simplification in our formal proof, because the Coq libraries provided too few theorems and tactics to work on the big integers.

This experiment also raises the question of *what do we perceive as a formally verified program?* The implementations described in this paper do run and produce output, however they need the whole context of the interactive theorem prover. We experimented with using the extraction facility of the Coq system to produce stand-alone programs that can be compiled with OCaml and run independently. This works, but the resulting program is one order magnitude slower than what runs in the interactive theorem prover. The reason is that the `BigZ` library exploits an ability to compute directly with machine integers (numbers modulo 2^{31}) [3], while the extracted program still views these numbers as records with 31 fields, with no shortcuts to exploit bit-level algorithmics. This raises several questions of trusted base: firstly, the Coq system with the ability to exploit machine integers directly for number computations has a wider trusted base (because the code linking integer computation with machine integer computation needs to be trusted). This first question is handled in another published article by Armand, Grégoire, Spiwack and Théry [3]. Secondly we also have to trust the implementation of the `native_compute` facility, which generates an `OCaml` program, calls the OCaml compiler, and then runs and exploits the results of the compiled program. This question is handled in another article by Boespflug, Dénès and Grégoire [10]. Thirdly, we could also extract the algorithms as modules to be interfaced with arbitrary libraries for large number computations. We would thus obtain implementa-

tions that would be partially verified and whose guarantees would depend on the correct implementation of the large number operations. This is probably the most sensible approach to using formally verified algorithms in the real world.

In the direction of formally verified programs, the next stage will be to study how the algorithms studied in this article can be implemented using imperative programming languages, avoiding stack operations and implementing clever memory operations, such as re-using explicitly the space of data that has become useless, instead of relying on a general purpose garbage-collector. Obviously, we would need to interface with a library for large number computations in such a setting. Such libraries already exist, but none of them have been formally verified. We believe that the community of formal verification will produce such a formally verified library for large number computations, probably exploiting the advances provided by the CompCert formally verified compiler [33] (which provides the precise language for the implementation), and the Why3 tool [20] to organize proofs of programs with imperative features, based on various forms of Hoare logic.

In their current implementation, our algorithms run at speeds that are several orders of magnitude lower than the same algorithms implemented by clever programmers in heavy duty libraries like `mpfr` [21]. For now, the algorithms for elementary operations are based on Karatsuba-like divide-and-conquer approaches, with binary tree implementations of large numbers, but it could be interesting to implement fast-Fourier-transform based multiplication as suggested by Schönhage and Strassen [38] and observe whether this brings a significative improvement in the computation of billions of decimals.

In spite of the fun with mathematical curiosities around the π number, the real lesson of this paper is more about the current progress in interactive theorem provers. How much mathematics can be described formally now? How much detail can we give about computations? How reproducible is this experiment?

For the question on how much mathematics, it is quite satisfying that real analysis becomes feasible, with concepts such as improper integrals, power series, interchange between limits, with automatic tools to check that mathematical expressions stay within bounds, but also with rigidities coming from the limits of the automatic tool. One of the rigidity that we experienced is the lack of a proper integration of square roots in the automatic tool that deals with equalities in a field. This tool, named `field`, deals very well with equalities between expressions that contain mostly products, divisions, additions and subtractions, but it won't simplify expressions such as $\sqrt{\frac{1}{\sqrt{2}}+\frac{1}{\sqrt{2}}}-\sqrt{\sqrt{2}}$. From a human user's perspective, this rigidity is often hard to accept, because once the properties of the square root function are understood, we integrate them directly in our mental calculation process.

For the question on how much detail we can give about computations, these experiments show that we can go quite far in the direction of reasoning about computation errors. This is not a novelty, and many other experiments by other

authors have been studying how to reason about floating point computations [12]. This experiment is slightly different in that it relies more on fixed point computations.

For the question on how reproducible is this experiment, we believe that one should distinguish between the task of running the formalized proof and the task of developing it. For the first task, re-running the formal proof, we provide a link to the sources of our developments [8], which can be run with Coq version 8.5 and 8.6 and precise versions of the libraries Coquelicot and Interval. For the task of developing the formal proof, this becomes a question at the edge of our scientific expertise, but still a question that is worth asking. In the long run, formally verified mathematics should become practical to a wider audience thanks to the availability of comprehensive and well-documented libraries such as Coquelicot [11] or mathematical components [22]. However, there are some aspects of the work that make reproducibility by less expert users difficult. For instance, it is often difficult to understand the true limits of automatic tools and this form of rigidity may cause users to lose a lot of time, for instance by mistaking a failure to prove a statement with the fact that the statement could be wrong. Another example is illustrated with the use of filters in the Coquelicot library, which requires much more advanced mathematical expertise than what would be expected for an intermediate level library about real analysis.

References

1. Gert Almkvist and Bruce Berndt. *Gauss, Landen, Ramanujan, the arithmetic-geometric mean, ellipses, π, and the Ladies Diary (1988)*, pages 125–150. Springer International Publishing, 2016.
2. Anonymous. Première composition de mathématiques", concours externe de recrutement de professeurs certifiés, section mathématiques, 1995.
3. Michaël Armand, Benjamin Grégoire, Arnaud Spiwack, and Laurent Théry. Extending Coq with Imperative Features and Its Application to SAT Verification. In *Interactive Theorem Proving, First International Conference, ITP 2010*, volume 6172 of *LNCS*, pages 83–98. Springer, 2010.
4. David Bailey, Peter Borwein, and Simon Plouffe. On the rapid computation of various polylogarithmic constants. *Mathematics of Computation*, 66(218):903–913, 1997.
5. Yves Bertot. Fixed precision patterns for the formal verification of mathematical constant approximations. In *Proceedings of the 2015 Conference on Certified Programs and Proofs*, pages 147–155. ACM, 2015.
6. Yves Bertot and Guillaume Allais. Views of Pi: definition and computation. *Journal of Formalized Reasoning*, 7(1):105–129, October 2014.
7. Yves Bertot, Nicolas Magaud, and Paul Zimmermann. A proof of GMP square root. *Journal of Automated Reasoning*, 29(3-4):225–252, 2002.
8. Yves Bertot, Laurence Rideau, and Laurent Théry. Distant decimals of pi, 2017. Available at `https://www-sop.inria.fr/marelle/distant-decimals-pi/`.
9. Frédéric Besson. Fast reflexive arithmetic tactics the linear case and beyond. In *Proceedings of the 2006 International Conference on Types for Proofs and Programs*, volume 4502 of *LNCS*, pages 48–62. Springer, 2007.
10. Mathieu Boespflug, Maxime Dénès, and Benjamin Grégoire. Full reduction at full throttle. In *Certified Programs and Proofs: First International Conference*, volume 7086 of *LNCS*, pages 362–377. Springer, 2011.

11. Sylvie Boldo, Catherine Lelay, and Guillaume Melquiond. Coquelicot: A user-friendly library of real analysis for Coq. *Mathematics in Computer Science*, 9(1):41–62, March 2015.
12. Sylvie Boldo and Guillaume Melquiond. Flocq: A unified library for proving floating-point algorithms in Coq. In *IEEE Symposium on Computer Arithmetic*, pages 243–252. IEEE Computer Society, 2011.
13. Jonathan M. Borwein and Peter B. Borwein. *Pi and the AGM: A Study in the Analytic Number Theory and Computational Complexity*. Wiley-Interscience, New York, NY, USA, 1987.
14. Richard P. Brent. Fast multiple-precision evaluation of elementary functions. *J. ACM*, 23(2):242–251, April 1976.
15. Henri Cartan. Théorie des filtres. *Comptes Rendus de l'Académie des Sciences*, 205:595–598, 1937.
16. Cyril Cohen, Maxime Dénès, and Anders Mörtberg. Refinements for free! In *Certified Programs and Proofs: Third International Conference*, volume 8307 of *LNCS*, pages 147–162. Springer, 2013.
17. Luis Cruz-Filipe. *Constructive Real Analysis: a Type-Theoretical Formalization and Applications*. PhD thesis, University of Nijmegen, April 2004.
18. Maxime Dénès, Anders Mörtberg, and Vincent Siles. A refinement-based approach to computational algebra in Coq. In *Interactive Theorem Proving - Third International Conference, ITP 2012*, volume 7406 of *LNCS*, pages 83–98. Springer, 2012.
19. Coq development team. The Coq proof assistant, 2016. `http://coq.inria.fr`.
20. Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In *Programming Languages and Systems: 22nd European Symposium on Programming, ESOP 2013*, volume 7792 of *LNCS*, pages 125–128. Springer, 2013.
21. Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. Mpfr: A multiple-precision binary floating-point library with correct rounding. *ACM Trans. Math. Softw.*, 33(2), June 2007.
22. Georges Gonthier et al. Mathematical components. Available at `http://math-comp.github.io/math-comp/`.
23. Boris Gourevitch. The world of Pi, 1999. Available at `http://www.pi314.net`, consulted March 2017.
24. Benjamin Grégoire and Laurent Théry. A purely functional library for modular arithmetic and its application to certifying large prime numbers. In *Automated Reasoning: Third International Joint Conference, IJCAR 2006*, volume 4130 of *LNCS*, pages 423–437. Springer, 2006.
25. Thomas C. Hales et al. A formal proof of the Kepler conjecture. *arXiv*, 1501.02155, 2015.
26. John Harrison. Formalizing basic complex analysis. In *From Insight to Proof: Festschrift in Honour of Andrzej Trybulec*, volume 10(23) of *Studies in Logic, Grammar and Rhetoric*, pages 151–165. University of Białystok, 2007. `http://mizar.org/trybulec65/`.
27. John Harrison. Pi series in Bailey/Borwein/Plouffe *polylogarithmic constants* paper, the HOL Light library, 2010. Available at `https://github.com/jrh13/hol-light/blob/master/Examples/polylog.ml`.
28. John Harrison. *Theorem Proving with the Real Numbers*. Springer Publishing Company, Incorporated, 1st edition, 2011.
29. Johannes Hölzl. Proving inequalities over reals with computation in Isabelle/HOL. In *Proceedings of the ACM SIGSAM 2009 International Workshop on Programming Languages for Mechanized Mathematics Systems (PLMMS'09)*, pages 38–45, Munich, 2009.
30. Johannes Hölzl, Fabian Immler, and Brian Huffman. Type classes and filters for mathematical analysis in Isabelle/HOL. In *Interactive Theorem Proving (ITP 2013)*, volume 7998 of *LNCS*, pages 279–294. Springer, 2013.
31. Louis V. King. *On the Direct Numerical Calculation of Elliptic Functions and Integrals*. Cambridge University Press, 1924.
32. Robbert Krebbers and Bas Spitters. Type classes for efficient exact real arithmetic in Coq. *Logical Methods in Computer Science*, 9(1), 2011.

33. Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, July 2009.
34. Érik Martin-Dorel and Guillaume Melquiond. Proving tight bounds on univariate expressions with elementary functions in Coq. *Journal of Automated Reasoning*, 57(3):187–217, 2016.
35. Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Springer, Berlin, Heidelberg, 2002.
36. Russell O'Connor. Certified exact transcendental real number computation in Coq. In *Theorem Proving in Higher Order Logics: 21st International Conference, TPHOLs 2008*, volume 5170 of *LNCS*, pages 246–261. Springer, 2008.
37. Eugene Salamin. Computation of π using arithmetic-geometric mean. *Mathematics of Computation*, 33(135), 1976.
38. Arnold Schönhage and Volker Strassen. Schnelle Multiplikation großer Zahlen. *Computing*, 7(3-4):281–292, 1971.
39. Alexey Solovyev and Thomas C. Hales. Formal verification of nonlinear inequalities with Taylor interval approximations. In *NASA Formal Methods: 5th International Symposium, NFM 2013,*, volume 7871 of *LNCS*, pages 383–397. Springer, 2013.